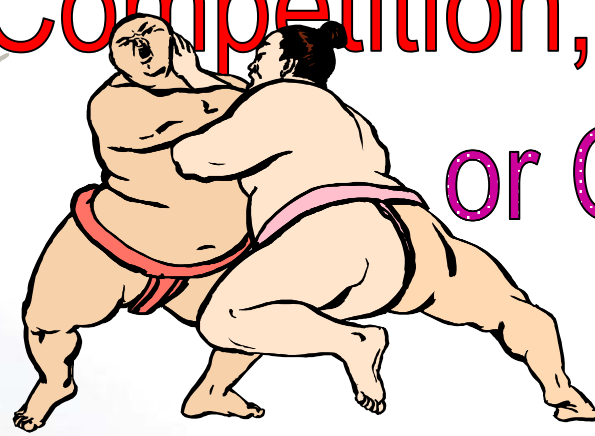
Ada and Real-Time Java™

Cooperation,

Competition,



or Cohabitation?

Ben Brosgol

Ada Core Technologies

brosgol@gnat.com ♦ www.gnat.com

SIGAda 2003 Conference

Introduction

Purpose of presentation

- Assess Java as a technology for real-time applications
  - Focus on thread model
- Summarize Real-Time Specification for Java (“RTSJ”)
- Compare Ada and RTSJ
  - Development process
  - Technology

Audience background

- Reasonable familiarity with Ada
- Some knowledge of Java
- Some knowledge of real-time issues
- No knowledge of the real-time Java proposals

Presenter credentials

- “Ada graybeard”
- Real-Time Java participant

Why Consider Java for Real-Time?

General benefits

- Language security and (in general) well-defined semantics
- Portability at multiple levels ("Write Once, Run Anywhere")
- Extensive API

Technical features / expressiveness / flexibility

- Support for software engineering (encapsulation, OOP, exceptions…)
- Built-in feature for concurrency (threads)
- Dynamic loading attractive in some segments such as telecom

Advantages over other languages

- Safer than C, simpler than C++, more popular than Ada

Pragmatics / politics

- Organization adopting Java as an "enterprise" language may be tempted to use Java for real-time

Summary of Java Thread Model (1)

Basic approach

- Extend `Thread` or implement `Runnable` and override `run()`
- Construct a `Thread` object `t` and invoke `t.start()`
- `sleep(millis)` suspends the calling thread
- `t.join()` suspends until the target thread `t` completes

Mutual exclusion

- `volatile` fields
- `synchronized` blocks/methods

Thread coordination/communication

- Pulsed signal: `obj.wait()` / `obj.notify()`
- Broadcast signal: `obj.wait()` / `obj.notifyAll()`

Summary of Java Thread Model (2)

Scheduling/priorities

- Priority is in range 1..10
- Thread can change or interrogate its own or another thread's priority
- `yield()` gives up the processor

Asynchrony

- `interrupt()` sets a bit that can be polled
- `suspend()` and `resume()` (deprecated)
- Asynchronous termination
  - `stop()` throws an asynchronous exception (deprecated)
  - `destroy()` kills a thread (unimplemented, on "endangered species" list)

Thread group

- Allows user to define method that is invoked when a thread dies from an unhandled exception

Critique of Java Thread Model (1)

Error-prone

- Requires cooperation by the accessing threads
  - Even if all methods are synchronized, an errant thread can access non-private fields without synchronization
- Subtle bug: constructor or synchronized instance method making non-synchronized access to static field
- “Nested monitor” problem

Subtleties in practice

- Not always clear when a method needs to be declared as synchronized
- Complex interactions with other features (e.g. when are locks released)
- Locking is hard to get right (exacerbated by absence of nested objects)

Effect not always clear from source syntax

- A non-synchronized method may be safe to invoke from multiple threads
- A synchronized method might not be safe to invoke from multiple threads

Critique of Java Thread Model (2)

Thread communication/synchronization issues

- `wait()` and `notify()`/`notifyAll()` are low-level constructs that must be used very carefully
  - "`while (!condition) {obj.wait()}`" needed
- Limited mechanisms for direct inter-thread communication
- Synchronized code that changes object's state must explicitly invoke `notify()` or `notifyAll()`
- No syntactic distinction between signatures of synchronized method that may suspend a caller and one that does not
- Only one wait set per object (versus per associated "condition")

Public thread interface issues

- The need to explicitly initiate a thread by invoking its `start()` method allows several kinds of programming errors
- Although `run()` is part of a thread class's public interface, invoking it explicitly is generally an error

Problems with Java for Real-Time (1)

Lack of some features useful for software engineering

- Operator overloading, strongly typed primitive types, …

Thread model deficiencies

- Priority range (1..10) too narrow
- Priority semantics are implementation dependent and fail to prevent unbounded priority inversion
- Relative `sleep()` not sufficient for periodicity

Memory management unpredictability

- Predictable, efficient garbage collection appropriate for real-time applications is not (yet) in the mainstream
- Java lacks stack-based objects (arrays and class instances)
- Heap used for exceptions thrown implicitly as an effect of other operations

Lack of features for accessing the underlying hardware

Regular Java Semantics for Scheduling

Section 17.12 of the Java Language Specification

- "Every thread has a *priority*. When there is competition for processing resources, threads with higher priority are *generally* executed in preference to threads with lower priority. *Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.*"

Problems for real-time applications

- Impossible to guarantee that deadlines will be met for periodic threads
  - May get priority inversion
- No guarantee that priority is used for selecting a thread to unblock when a lock is released
- No guarantee that priority is used for selecting which thread is awakened by a `notify()`, or which thread awakened by `notifyAll()` is selected to run

Problems with Java for Real-Time (2)

Asynchrony deficiencies

- Event handling requires dedicated thread
- `interrupt()` not sufficient
- `stop()` and `destroy()` deprecated or dangerous

Run-time issues

- Dynamic class loading is expensive, not easy to see when it will occur
- Array initializers ⇨ run-time code

OOP has not been embraced by the real-time community

- Dynamic binding complicates analyzability
- Garbage Collection defeats predictability

Performance questions

“Standard” API would need to be rewritten for predictability

Some JVM opcodes require non-constant amount of time

History of Real-Time Java Efforts

1998

NIST Workshops
Lisa Carnahan, NIST

Jan 1999

Sun JCP: JSR-001
Real-Time for Java Expert Group
Greg Bollella (IBM/Sun)
www.rtj.org

J-Consortium
Real-Time Java WG
Kelvin Nilsen (NewMonics / Aonix)
www.j-consortium.org

Jun 2000

Real-Time Specification for Java

Real-Time Core Extensions

Nov 2001

RTSJ V1.0: RI, TCK
Doug Locke, Peter Dibble
(Timesys)

JEFF Standard

July 2003

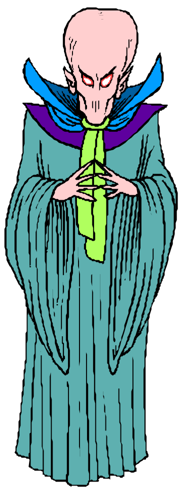
Merge into common spec?
The Open Group

Early 2004

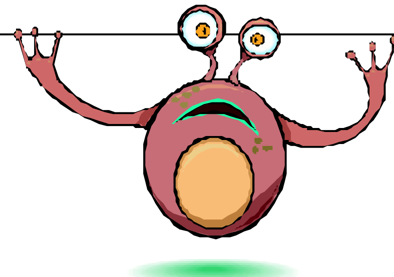
RTSJ V1.0.1

Focus of this presentation will be on the Real-Time Specification for Java

al-Time for Java™ Expert Gro

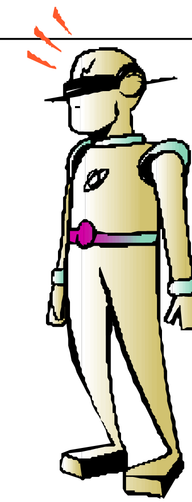


James Gosling
Sun

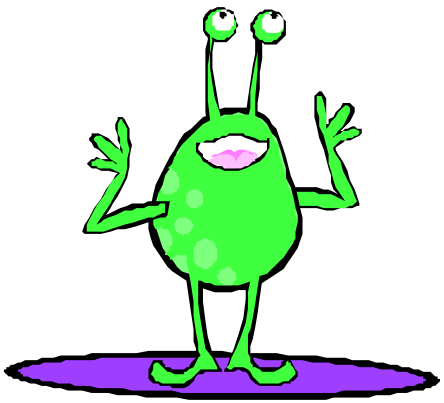


Greg Bollella
IBM → Sun

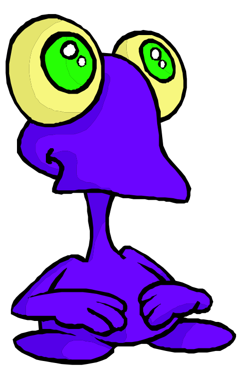
Steve Furr
QSSL



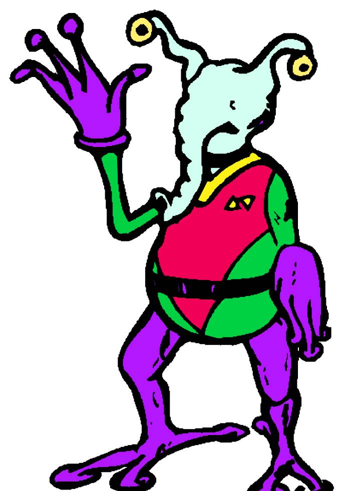
Dave Hardin
Rockwell-Collins →
aJile Systems



Ben Brosgol
Aonix → Ada Core



Peter Dibble
Microware → Timesys



Mark Turnbull
Nortel Networks

Paul Bowman
Cyberonics

Summary of Main RTSJ Features

Concurrency

- Class `RealtimeThread` extends `java.lang.Thread`
- Flexible scheduling framework together with default scheduler
- Several mechanisms for priority inversion avoidance

Memory Areas

- Immortal, Scoped Memory augment Garbage-Collected Heap
- “NoHeap Realtime Thread” can preempt GC

Asynchrony

- Asynchronous Event Handling
- Asynchronous Transfer of Control

Time and Timers

Low-Level Features

- Specialized kinds of “physical” memory
- “Peek/poke” of primitive data in “raw” memory

Scheduling in the RTSJ

General concept of "schedulable object"

- Realtime thread or asynchronous event handler
- Arguments to constructor establish scheduling characteristics (e.g. priority) and release characteristics (e.g. cost, periodicity)

Initial default scheduler

- Must support at least 28 distinct priority values, beyond Java's 10
- Preemptive, fixed priority, FIFO within priority

Support for feasibility analysis (optional)

- Implementation can query release parameters to determine if a set of schedulable objects can satisfy some constraint

Flexibility

- Implementation can install arbitrary scheduling algorithms
- Users can replace these dynamically, can have different schedulers for different schedulable objects

Synchronization in the RTSJ

Monitor control policy allows user to select which policy governs which objects

- Semantics defined for default scheduler
- Distinction between active and base priority

Priority Inheritance is default policy

- May be changed by user at system startup

Priority Ceiling Emulation is also defined (but is optional)

- Locking thread's priority is boosted to ceiling when lock acquired, reset when lock released
- Ceiling violation exception thrown if locking thread has higher priority than the ceiling
- No requirement for non-blocking as in Ada

"Wait-free queues" allow communication between a NoHeap Realtime Thread and a regular Java thread

Memory Areas in the RTSJ

Goals

- Augment heap with areas not subject to Garbage Collection
- Do not compromise Java safety (i.e., no explicit “free”)

Heap

- Subject to Garbage Collection

Immortal Memory

- Not subject to GC, never reclaimed
- May reference the heap and vice versa

Scoped Memory

- Transient stack-like area, not subject to GC
- May reference heap, immortal, outer scoped areas, but not vice versa
- Assignment rules prevent dangling references
- Reference count scheme establishes when scoped area is freed

Asynchrony in the RTSJ

Asynchronous Event Handler

- Use for hardware interrupts or software "happenings"
- An AEH is a schedulable object but need not have a dedicated thread
- Override a method to implement the relevant event handling
- Associate one or more Asynch Event Handlers with an Asynch Event
  - Firing an AE → schedule associated AEHs

Asynchronous Transfer of Control ("ATC")

- Use for timing out on a computation, aborting a thread
- Methodologically questionable, and complicated to implement
  - Conflict between desire for ATC to be immediate, and the need for certain code to execute completely
- Extends `t.interrupt()` to real-time threads, throwing an exception not only when `t` is blocked but also when `t` is executing *asynchronously interruptible* ("AI") code
  - Synchronized code, and methods lacking a special throws clause, are not AI

Ada and RTSJ - The Process

Ada

- Sponsored “top down” effort ⇨ ISO standard + Rationale
- Detailed audit trail (LSNs, AIs, etc.)
- Thorough review (ARG, WG9)
- Highly open process (public briefings, etc.)
- Product evolution based on ISO rules

RTSJ

- Focused “bottom up” volunteer effort ⇨ *de facto* standard
- JCP requires not just the spec but also a RI and TCK
- Audit trail comprises principally the group’s e-mail messages
- Review was principally internal in RTJEG
- Semi-open process
- Product evolution based on Sun’s JCP rules

Ada and RTSJ: The Technology

Ada

- ☺ Performance (classical stack-based language, queueless lock management)
- ☺ Conservatism (traditional static compile/bind/link)
- ☺ Well-defined semantics (queue placement)
- ☺ Cleaner / simpler approach to ATC
- ☺ Existence of good implementations now
- ☺ Allows but does not require OOP paradigm
- ☹ Market perception

RTSJ

- ☺ Flexibility (multiple schedulers, dynamic loading...)
- ☺ Functionality (RationalTime class, feasibility)
- ☹ Style may seem complicated to traditional Java programmers
  - Need to pay attention to memory management issues
- ☹ Performance questions

How Can Ada Experience Help Real-Time Java?

Specific technical ideas may be borrowed/adapted

From BMB presentation
To RTJEG, March '99

- Absolute delay (sleepUntil method)
- Scheduling policies
- Concept of "abort-deferred" regions of code
- Priority ceilings for efficient lock management
- Subsets for specialized application areas

Political lessons

- Remember that customers want solutions, not technology
- Beware the culture clash
  - Real-time applications take a static approach to ensure predictability
  - All heap objects are allocated at system startup
  - OOP and garbage collection have not been popular

Challenges

- Sacrificing performance/flexibility for safety (an effect of Garbage Collection) has always been a hard sell to the real-time community

Ada and Real-Time Java: Friends or Foes?

Friends

- “The enemy of my enemy is my friend”
- Cross-fertilization of ideas beneficial to both
  - Many Ada concepts influenced RTSJ and Real-Time Core Extensions
    - Priority Ceiling, ATC, absolute delay, Ravenscar profile
  - RTSJ can serve as model for future Ada work in some areas
    - “On line” feasibility analysis, integrated support for real-time characteristics
- RTSJ-compliant JVM is feasible target for Ada

Foes

- Ada and Real-Time Core Extensions compete in same market
  - But RTCE has not yet been implemented

Peaceful coexistence

- Ada and RTSJ have different markets
  - Ada: traditional real-time
  - RTSJ: organization already committed to Java