

# Chapter 8



## Introduction to the Real-Time Java Platform

- ▼ A BRIEF HISTORY OF REAL-TIME JAVA
- ▼ MAJOR FEATURES OF THE SPECIFICATION
- ▼ IMPLEMENTATION
- ▼ RTSJ HELLO WORLD

**A** real-time specification for Java is a rather remarkable idea. Java programs running on a JVM are, as a rule, much slower than similar programs written in C and compiled to the target processor. Even worse for real time, garbage collection stops everything else from time to time. These are not the characteristics of a good real-time platform. The promise of the Java platform for real time is that Java specifies a complete platform, and the charter of the Real Time Java Expert Group allowed it a free hand with the entire scope of the Java platform.<sup>1</sup> The Java platform includes aspects of the system from a robust class library and language specification all the way down to the instruction set of the virtual processor and many details of the multitasking runtime.

1. The permission to change Java was not as open as it sounds. As James Gosling pointed out at an early Expert Group meeting, if we crossed some undefined line in our changes, it would not matter what we called it. It would not be Java.



Previous attempts at real-time standards have struggled with limited scope. The most significant previous effort was the POSIX real-time specification. That specification had to be UNIX-like, and it could specify only a library API. Its control of the language was limited to “calling out” the ANSI C specification, and POSIX says nothing about the underlying machine’s instruction set.

## A Brief History of Real-Time Java

Many people contributed to the idea and its realization. Gallons of rhetorical blood and sweat were invested in a brief specification. Kelvin Nilsen deserves credit for starting the process. Years before Java appeared in public, Kelvin invented a garbage collection algorithm with good real-time characteristics. First, he embedded most of his garbage collection system in hardware, then he built software-only implementations. Kelvin promoted his idea at academic conferences and in the marketplace, but it did not catch on. One problem may have been that it required a specially instrumented compiler. Java was just what Kelvin needed, a new language with no legacy code, which generated interpreted byte-code and which needed garbage collection. He could fit his garbage collector into the JVM, and the JVM needed an improved garbage collector. Kelvin started a real-time Java working group to discuss ways to improve Java’s real-time characteristics.

IBM and Sun were also interested in a real-time version of Java. They started efforts to build interest in real-time Java about the same time Kelvin did. It all came together in a grand meeting where Sun, IBM, and NIST (National Institute of Standards and Technology) jointly blessed a working group. It commenced meeting under the aegis of NIST and the leadership of Lisa Carnahan. After several months of meetings, the group produced a document called *Requirements for Real-Time Extensions for the Java™ Platform: Report from the Requirements Group for Real-Time Extensions for the Java™ Platform*. The report lists 53 groups as joint authors.

Around September of 1998, Sun announced the Java Community Process, a new process for maintaining and extending the Java specification. IBM promptly submitted a request for a real-time Java specification based partly on the NIST requirements document. The request, the first Java Specification Request (JSR-000001), was accepted in December of 1998.

Greg Bollella from IBM was selected as the JSR-000001 Specification Lead, and he formed an Expert Group with two tiers to help him create the specifica-



tion. The primary group would do most of the work. Table 8–1 lists the members of the Expert Group.

**Table 8–1** Real-Time Specification for Java Primary Expert Group

<b>Greg Bollella</b>	IBM
<b>Paul Bowman</b>	Cyberonics
<b>Ben Brosgol</b>	Aonix/Ada Core Technologies
<b>Peter Dibble</b>	Microware Systems Corporation/ TimeSys
<b>Steve Furr</b>	QNX System Software Lab
<b>James Gosling</b>	Sun Microsystems
<b>David Hardin</b>	Rockwell-Collins/ajfile
<b>Mark Turnbull</b>	Nortel Networks

The Consultant Group would provide advice and participate in the major iterations of the specification. Table 8–2 lists the members of the Consultant Group.

**Table 8–2** Real-Time Specification for Java Consultant Group

<b>Rudy Belliardi</b>	Schneider Automation
<b>Alden Dima</b>	National Institute of Standards and Technology
<b>E. Douglas Jensen</b>	MITRE
<b>Alexander Katz</b>	NSICom
<b>Masahiro Kuroda</b>	Mitsubishi Electric
<b>C. Douglass Locke</b>	Lockheed Martin/TimeSys
<b>George Malek</b>	Apogee
<b>Jean-Christophe Mielnik</b>	Thomson-CSF
<b>Ragunathan Rajkumar</b>	CMU
<b>Mike Schuette</b>	Motorola
<b>Chris Yurkoski</b>	Lucent
<b>Simon Waddington</b>	Wind River Systems

The combined Expert Groups first met at the Spring 1999 Embedded Systems Conference and started serious work in March 1999.

In September of 1999, the specification was published for “participant review.” This is a formal stage in the Java Community Process in which the Expert Group shows a preliminary specification to other people who are involved in the process. In this case, the Expert Group decided to publish the specification on an



## *Real-Time Java Platform Programming*

open Web site. Formally, it was a participant review, but the document was visible to the world. Comments came in and the specification was improved. The official public review stage started in December 1999. More comments arrived and the specification was further improved. Finally, after about a year of steady work, the Expert Group released the preliminary edition of The Real-Time Specification for Java, which was printed and ready to be distributed in June 2000 at JavaOne.

The first edition of the specification was not an official specification. The Java Community Process requires three things before a specification is accepted: the specification, a reference implementation, and a test suite. Not only are the reference implementation and test suite required before anyone can write products that claim conformance, they also serve to prove that the specification can be implemented and is generally sane. The specification book was published before the other tasks were complete, to make it readily available to people tracking the standard and to draw more public interest and comment.

Through 2000 and most of 2001, the Expert Group continued to meet in frequent conference calls. Late in 2000, TimeSys volunteered to create the reference implementation, and they delivered a preliminary reference implementation to the group in April 2001. Naturally, a usable implementation of the preliminary specification focused attention on some areas that needed improvement. The sections of the specification on asynchronous transfer of control and scoped memory, in particular, were carefully studied. A revised specification, a reference implementation that conformed to the revised specification, and a test suite were submitted to the JCP Executive Committee for approval in October 2001.

### **Major Features of the Specification**

The Real-Time Specification for Java enhances the Java specification in six ways:

- 1.** It adds real-time threads. These threads have scheduling attributes that are more carefully defined than is the scheduling for ordinary Java threads.
- 2.** It adds tools and mechanisms that help programmers write Java code that does not need garbage collection.
- 3.** It adds an asynchronous event handler class and a mechanism that associates asynchronous events with happenings outside the JVM.
- 4.** It adds a mechanism called asynchronous transfer of control that lets a thread change control flow in another thread. It is, essentially, a carefully controlled way for one thread to throw an exception into another thread.
- 5.** It adds mechanisms that let the programmer control where objects will be allocated in memory.



6. It adds a mechanism that lets the programmer access memory at particular addresses.

What the Real-Time Java does not change may be as important as the things it changes. Ordinary Java programs will run on an implementation of the Real-Time Specification. They can even run while the JVM is executing real-time code. There is no magic that will cause ordinary Java programs to become more timely when they run on a JVM that implements the Real-Time Specification, but they won't behave any worse than they did.

Furthermore, non-real-time code will not interfere with real-time code unless they share resources.

### **Threads and Scheduling**

Whether it is by priority scheduling, periodic scheduling, or deadline scheduling, the way tasks are scheduled on the processor is central to real-time computing. Non-real-time environments (like a standard JVM) can be casual about the details of scheduling, but a real-time environment must be much more precise. The specification treads a line between being specific enough to let designers reason about the way things will run but still flexible enough to permit innovative implementation of the RTSJ. For instance, there is only one method for which the RTSJ requires every implementation to meet a particular performance goal: allocation from an `LTMemory` area.

#### ***LTMemory Performance***

The **RTSJ** specifies a high standard for the performance of `LTMemory` allocation because that allocation mechanism is intended for use in the tightest time-critical code. The specification is trying to assure designers that allocation of `LTMemory` is safe for critical real-time code.

Allocation from an `LTMemory` area must need time that is linear in the size of the allocation memory. That is the best possible allocation performance. Memory allocation in the JVM has several stages: first the right amount of free memory is allocated, then the memory is initialized in various stages under the control of the JVM and the class constructors. Every field in the object must be initialized before that field is used. In some cases, some initialization can be deferred, but ultimately every byte in the object is initialized.

The implementor can use any allocation algorithm that has asymptotically better performance than initialization of the allocated memory.

The **RTSJ** includes priority scheduling because it is almost universally used in commercial real-time systems and because all legacy Java applications use priority scheduling. The **RTSJ** requires at least 28 real-time priorities in addition to the



## Real-Time Java Platform Programming

10 priorities called for by the normal JVM specification. The RTSJ calls for strict fixed-priority preemptive scheduling of those real-time priorities. That means that a lower-priority thread must never run when a higher-priority thread is ready. The RTSJ also requires the priority inheritance protocol as the default for locks between real-time threads, permits priority ceiling emulation protocol for those situations, and provides a hook for other protocols.

The RTSJ provides room for implementors to support other schedulers. The specification does not define the way new schedulers will be integrated with the system; it only says that an implementor may provide alternate schedulers and defines scheduler APIs that are general enough to support a wide variety of scheduling algorithms.

### **Socket Scheduling**

While the Expert Group was refining the scheduling interfaces, we, to prevent ourselves from designing interfaces that would only accommodate known schedulers, invented a series of sock schedulers that would schedule according to various properties of socks.

### **Garbage Collection**

The standard JVM specification does not require garbage collection. It requires dynamic memory allocation and has no mechanism for freeing memory, but the Java Language Specification does not require any particular solution for this massive memory leak. Almost every JVM has a garbage collector, but it is not required.

### **GC-less JVM**

David Hardin (on the Expert Group) had extensive experience with the Rockwell Collins JEM chip. It is a hardware implementation of Java and has no garbage collector. Programming with no garbage collector requires discipline, and many standard Java idioms become convoluted, but it works well enough that the JEM has become a modestly successful Java platform.

The RTSJ continues the policy of the original Java specification. The RTSJ discusses interactions with a garbage collector at length, but a Java runtime with no garbage collector could meet the specification.

The RTSJ, although it does not require a garbage collector, specifies at least one API that provides for a particular class of garbage collection algorithm. Incremental garbage collectors that pace their operation to the rate at which threads



create garbage are promising for real-time systems. Garbage collection can be scheduled as an overhead charge on the threads that create garbage and execute in brief intervals that do not disrupt other activities. The RTSJ has a constructor for real-time threads; the constructor includes a memory-parameters argument that can specify the allocation rate the garbage collector and scheduler should expect from the thread.

The Expert Group did not feel comfortable requiring a magical garbage collector and relying on it to make all the real-time problems with garbage collection disappear. Instead, we took the attitude that even the best-behaved garbage collector may sometimes be more trouble than it is worth to the real-time programmer. An implementation can provide any (correct) garbage collection algorithm it likes, and users will certainly appreciate a good one, but for real-time programming, the RTSJ provides ways to write Java code that will never be delayed by garbage collection.

The first tool for avoiding garbage collection is no-heap, real-time threads. These threads are not allowed to access memory in the heap. Since there is no interaction between no-heap threads and garbage collection or compaction, no-heap threads can preempt the garbage collector without waiting for the garbage collector to reach a consistent state. Ordinary threads and heap-using, real-time threads can be delayed by garbage collection when they create objects in the heap, and they have to wait for the garbage collector to reach a consistent state if they are activated while the garbage collector is running. No-heap, real-time threads are protected from these timing problems.

### ***Asynchronous Event Handlers***

Many real-time systems are event driven. Things happen and the system responds to them. It is easy to code an event-driven system structured so that each event is serviced by a thread created for that particular event, and it makes the scheduling attributes of each event clear to the scheduler. The idea sounds obvious. Why isn't it common practice? The time between an event and the service of the event is overhead on real-time responsiveness. Thread creation is slow. It is a resource-allocation service, and real-time programmers avoid resource allocation when they are concerned about time.

Asynchronous event handlers are an attempt to capture the advantages of creating threads to service events without taking the performance penalty.

Event-driven programming needs events. The standard Java platform has extensive mechanisms for input from its GUI, but no general-purpose mechanism for associating things that happen outside the Java environment with method



## Real-Time Java Platform Programming

invocation inside the environment. The RTSJ introduces *happenings* as a pathway between events outside the Java platform and asynchronous event handlers.

### **Asynchronous Transfer of Control**

Asynchronous transfer of control was a late addition to the RTSJ, and it was much harder to invent than you might think.

Asynchronous transfer of control (ATC) is a mechanism that lets a thread throw an exception into another thread. Standard Java includes a similar mechanism, `thread.interrupt`, but it is weak.

Why is ATC so important?

1. It is a way to cancel a thread in a forcible but controlled way.
2. It is a way to break a thread out of a loop without requiring the thread to poll a “terminate me” variable.
3. It is a general-purpose timeout mechanism.
4. It lets sophisticated runtimes take scheduler-like control of execution. People interested in distributed real time have powerful requirements for this control.

Why is ATC so hard?

1. Code that is not written to be interrupted may break badly if the JVM suddenly jumps out of it.
2. You cannot just jump from the current point of execution to the “right” catch block. The platform has to unwind execution through catches and, finally, clauses in uninterruptible methods until it has fully serviced the exception.
3. Nested methods may be waiting for different asynchronous exceptions. The runtime has to make certain that the exceptions get to the right catch blocks.

### **Memory Allocation**

By itself, support for no-heap, real-time threads would be useless. The thread would be restricted to elementary data types. It would not even be able to access its own thread object. The RTSJ created two new memory allocation domains to give no-heap threads access to objects: immortal memory and scoped memory.

Immortal memory is never garbage collected and would make no-heap threads thoroughly usable even without scoped memory. Immortal memory fits the large class of real-time programs that allocate all their resources in an initialization phase and then run forever without allocating or freeing any resources. Even systems written in C and assembly language use this paradigm. Even without garbage collection, resource allocation often has tricky timing characteristics



and nasty failure modes. It makes sense to move it out of the time-critical part of an application.

Immortal memory is simple to explain and implement, but it leads to unnatural use of the Java language:

- The Java Platform does not encourage reuse of objects. In some cases, properties of objects can only be set by their constructor, and the Java language's strong typing makes it impossible to reuse an object as anything other than exactly its original type. (The Java language has no union.)
- The Java class libraries freely create objects. A programmer who called innocuous methods in the collections classes or the math classes could quickly find immortal memory overflowing with throwaway objects created in the class libraries. Real-time code is not compelled to use standard class libraries, but those class libraries are a major attraction of Java and the effort involved in recoding them all to real-time standards would be staggering.

Scoped memory isn't as simple as immortal memory, but it goes a long way toward addressing the problems with immortal memory. In simple applications, scoped memory works like a stack for objects. When the thread enters a memory scope, it starts allocating objects from that scope. It continues allocating objects there until it enters a nested scope or exits from the scope. After the thread exits the scope, it can no longer access objects allocated there and the JVM is free to recover the memory used there.

If a thread enters a scope before calling a method in a standard class library and leaves the scope shortly after returning, all objects allocated by the method will be contained in the scope and freed when the thread leaves the scope.<sup>2</sup> Programmers can safely use convenience objects by enclosing the object creation and use in a scope. The mechanism (called a closure) for using scopes is a little ungainly, but an RTSJ programmer uses closures so much that they soon feel natural.

Performance is the most important cost of immortal and scoped memory. The RTSJ has access rules for no-heap, real-time threads and rules that govern the storage of references to objects in heap and scoped memory. These rules must be enforced by the class verifier or the execution engine. Unfortunately, it seems likely that the execution of the bytecodes that store references will have to do some part of that work. That necessity will hurt the JVM's performance.

2. Using standard libraries in scoped memory is not always as simple as it sounds. The most likely problem is that the library method will get a runtime exception when it tries to access heap memory.



### **Memory Access**

Special types of memory, I/O devices that can be accessed with load and store operations, and communication with other tasks through shared memory are important issues for embedded systems. It takes a bit of a stretch to call these real-time issues, but the RTSJ makes that stretch.

Special types of memory are closely related to performance (slow memory, cached memory, high-speed nonsnooped access to sharable memory, etc.) Whereas performance is not a real-time issue in the strictest sense, predictable performance *is* a real-time issue and some memory attributes like cacheable, sharable, and pageable have a large impact on the predictability of code that uses the memory.

The RTSJ “raw” memory access classes give something like “peek and poke” access to memory. They run through the JVM’s security and protection mechanisms, so this introduction of pointer-like objects does not compromise the integrity of the Java platform, but it does give enough direct access to memory to support device drivers written in Java and Java programs that share memory with other tasks.

The raw memory classes do nothing to improve the real-time performance of the Java platform. They are there because some of the most enthusiastic early supporters of a real-time Java specification wanted to use Java to write device drivers. It was a painless addition to the specification and it greatly increases the usefulness of Java for the embedded real-time community.

### **Implementation**

An implementation of the RTSJ must include the `javax.realtime` packages. It will almost certainly include a special JVM, and it might include a compiler that can help find memory reference errors.

Any compiler can make classes that use the RTSJ, and special classes on an ordinary Java platform can pretend to conform to the RTSJ. Those were important design goals. We wanted RTSJ users to be able to do a lot of development on standard IDEs.

Class libraries can implement the RTSJ APIs and provide some of the RTSJ functions, but they cannot provide all the services. A true RTSJ implementation includes enhancements to the JVM. There are no new bytecodes, but it would be difficult or impossible to enforce memory access rules without enhancing the implementation of the bytecodes that handle references. Asynchronous transfer of control also seems to require changes to bytecode interpretation, this time to the



bytecodes that move between methods. The RTSJ also adds new priorities and far stricter scheduling rules than those of the standard JVM.

A special RTSJ compiler would be useful. A compiler cannot always identify memory reference violations, but it can find enough of them to justify trying. It would be far better to find a reference violation as a compile-time error than as a runtime exception.

### RTSJ Hello World

Working with the TimeSys reference implementation of the RTSJ installed on a Linux system in `/usr/local/timesys` and with the Sun JDK tools, we show here a step-by-step procedure for creating and running a hello world program for real time.

Hello world doesn't have significant timeliness constraints, so Example 8-1 will just run the standard hello world in a real-time thread.

#### Example 8-1 RT hello world program

```
import javax.realtime.*;

public class Hello1 {
    public static void main(String [] args){
        RealtimeThread rt= new RealtimeThread(){
            public void run() {
                System.out.println("Hello RT world");
            }
        };
        if(!rt.getScheduler().isFeasible())
            System.out.println("Printing hello is not feasible");
        else
            rt.start();
    }
}
```

The program shown in Example 8-1 is a complete program. Since it prints "Hello RT world" from a real-time thread, that part of its execution can take advantage of priority inheritance and strictly defined, fixed-priority, preemptive scheduling. Those scheduling properties do not make much difference to the output of one short string, but they are there.



## Real-Time Java Platform Programming

To test the program:

1. If you are developing from the command line, you might use a command like this to compile Hello1:

```
javac -classpath /usr/local/timesys/rtsj-ri/lib/foundation.jar Hello1.java
```

2. If the real-time JVM is on your execution path and the classpath is set to your real-time Java class libraries, a simple command line will run the program:

```
tjvm Hello1
```

More likely, you'll need a command line like this:

```
tjvm -Djava.class.path=/home/dibble/javaprogs/hello1  
-Xbootclasspath=/usr/local/timesys/rtsj-ri/lib/foundation.jar Hello1
```

3. The output will be:

```
Hello RT world!
```