

Real-Time Java Memory Management Issues

Torbjörn Ekman, Anders Nilsson, Klas Nilsson, Roger Henriksson, Anders Ive
Sven Gestegård Robertz and Anders Blomdell
Lund Institute of Technology

6 June 2001

Abstract

The Java programming language and execution environment has gained a lot of interest among developers of embedded real-time systems in the last few years. There are numerous reasons for using Java in real-time systems, such as:

- Object orientation
- Strict type checking
- Precise exception handling
- Automatic memory management
- “Write once, run anywhere“

All of which ensures safety and improves programming productivity.

Current versions of Java as provided by Sun lack real-time properties, but the National Institute of Standards and Technologies has proposed real-time extensions requirements [req99]. Based on these requirements two standard proposals have emerged, the Real-Time Specification for Java [BGB⁺00] and Real-Time Core Extensions [cor00]. Both target areas such as scheduling and dispatching of threads, deterministic synchronization mechanisms, safe thread termination, and asynchronous transfer of control. The proposals are not limited to the current scheduling mechanisms but are extendible with new policies.

Both proposals assume that garbage collection is not deterministic and hence not suitable for hard real-time requirements. Therefore they introduce new memory handling mechanisms with, in their opinion, better predictability. However, an investigation of the chosen strategies, and their implications convinced us that both proposals' memory management strategies should be abandoned.

1 The Real-Time Specification for Java

The specification includes a class hierarchy of seven subclasses of the abstract class `MemoryArea` which represents memory. One of these classes is the traditional heap, of which a singleton instance exists. Each thread is assigned a default memory area from which objects are allocated, but the programmer can explicitly allocate objects from other memory areas as well. Some of the defined memory areas are scoped memory regions with a limited lifetime. At the end of the lifetime of a scoped memory area all objects residing in that area are reclaimed. Scoped memory areas may also be nested.

Because of different lifetimes for various memory areas, strict assignment rules to maintain pointer safety are necessary. Violation of these rules causes run-time exceptions. To achieve deterministic behavior for hard real-time threads, instances of the thread class `NoHeapRealTimeThread` may not have reachable memory areas where objects are relocated or garbage collected. These checks and validation of assignments must be performed at run-time, although some could possibly be performed through static code analysis. The specification also includes primitives and memory areas to access physical memory addresses.

2 Real-Time Core Extensions

The specification partitions the memory in a baseline heap and core memory. Objects in core memory are real-time objects and may not access objects on the garbage collected baseline heap. Objects on the baseline heap may, with certain restric-

tions, access core objects by invoking explicitly declared Core-Baseline methods on objects residing in core memory.

Each active object is assigned a default allocation context but objects can explicitly be allocated in other contexts as well. When the lifetime of an allocation context has passed, all objects in that context are eligible for reclamation. Because there might still be references from the baseline heap to those objects, the baseline garbage collector is responsible for deleting the objects not reachable from the baseline heap. When all objects have been deleted the entire allocation context is reclaimed.

There are no limitations on which allocation contexts objects may reference. It is the responsibility of the programmer not to release an allocation context as long as there are pointers referencing it.

The specification also allows stack allocation of objects. These objects are placed on the run-time stack and are automatically reclaimed as the scope is exited. To allocate stack objects, a set of restrictions apply and the reference must explicitly be declared stackable. Physical memory may be accessed through ports. These ports gives the programmer the possibility to directly read and modify the specified hardware address.

3 Memory Management Issues

Both proposals solve the problem with deterministic garbage collection by simply turning it off for hard real-time threads, leaving the memory management completely to the developer. They also introduce various restrictions for objects residing in different memory regions. Allowing direct access to physical memory limits the possibility to execute the same code both embedded and in simulated environments.

By introducing new memory types, and delegating the decision on which memory type to use to the programmer, there will always be a risk of introducing bugs related to manual memory management. So, while still maintaining the safety of Java, there are a number of drawbacks:

- Handling different memory areas complicates programming.
- Exceptions are likely to occur due to programming errors.

- The same program cannot be run in both embedded and simulated/virtual systems, which is a big disadvantage for, for instance, automation software.

These difficulties all stem from the assumption that real-time garbage collection cannot and should not (due to compatibility with existing kernels and threads) be used. In the proposed specifications we are then still left with much of the complexity of memory management in real-time programming that we wanted to get rid of, although in a different form.

Within our research, it has been demonstrated that predictable garbage collection, even in hard real-time, can be accomplished by the combination of incremental algorithms and scheduling of the collection work [Hen98]. The imposed overhead for automatic memory management does not affect the response times for high priority real-time threads nor allocation costs.

We currently have three projects exploring and refining different strategies for real-time garbage collection in embedded Java environments. Inspired by preliminary results, we claim that the current specification efforts concerning real-time extensions to Java are on the wrong track.

References

- [BGB⁺00] Greg Bollella, James Gosling, Benjamin M. Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-time Specification for Java*. Addison-Wesley, 2000.
- [cor00] Real-Time Core Extensions. International J Consortium Specification, J Consortium, 2000.
- [Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, 1998.
- [req99] Requirements for Real-time Extensions for the Java Platform. NIST Special Publication 500-243, National Institute of Standards and Technology, 1999.