Java in Safety Critical Systems

Andy Walter
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany

Dr. James J. Hunt
aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany

April 13, 2010

1 Abstract

Until recently, the preferred language for developing safety critical applications has been Ada, but this is
beginning to change. The number of developers willing to program in Ada is diminishing, while the complexity
of applications is increasing. Where as C and C++ are poor alternatives to Ada, realtime Java specifications
have benefited from strong cross fertilisation from the Ada community, giving realtime Java most of Ada’s
advantages for developing safety critical systems.

Though strongly related to standard Java technology such as J2SE and J2EE, realtime Java is really a
different beast. The differences are subtle, so as to benefit from a common language base; but essential.
Realtime Java sets itself apart by having much stronger threading semantics: it provides a strict specification
of thread priorities and protocols for avoiding Priority Inversion. The RTSJ also introduces techniques for
avoiding timing anomalies caused by garbage collection, ideally while maintaining the reference consistency
automatic object deallocation ensures.

In the past, reference consistency in safety critical applications was maintained by disallowing or severely
limiting dynamic memory management. This approach works well for state machine like tasks, but not for
more complex applications. Many complex safety critical applications use object pooling to dynamically
manage memory to get around this ad hoc restriction. The emerging Safety Critical Java standard (JSR 302)
provides more flexibility than currently tolerated by providing a stack like approach to memory allocation and
deallocation. This will enable the Java language to be used at the highest criticality levels in the near term,
but it does not address increasing complexity well.

In the long term and for applications into the medium criticality in the near term, where complexity is
already challenging, realtime garbage collection offers a more practical solution. Garbage collection relieves
the application developer of reference inconsistency concerns, such as dangling pointers and memory leaks,
since these can be guaranteed by the Java runtime environment. A deterministic, realtime garbage collector can
also ensure that the garbage collection process does not interfere with applications meeting timing deadlines.

Current work on object oriented technology in SG-5 of the SC-205/WG-71 Plenary to update the DO-178
standards, will make certification of Java technology, including the use of virtual machine technology and
garbage collection, easier. In the past, these technologies were up to the discretion of individual certification
experts, who often have only minimal understanding of OO Technology. New standards will provide both
stronger rules and rationale for how certification should be conducted.

This paper outlines the important Java standards, such as the realtime Specification for Java (JSR 1 and
JSR 282) and Safety Critical Java (JSR 302), as well as proposed changes from SG-5 for object oriented
technology. New garbage collection technology will also be covered. This should give the attendee a good
background in the state-of-the-art of realtime Java Technology and safety certification.

2 The Realtime Specification for Java (RTSJ)

The Realtime Specification for Java[2], which was published by the Java Community in 2001 as JSR 1[4],
has been widely accepted as the standard library extension for realtime Java Virtual Machines and is used by
most of the realtime Java applications today. Currently, the Java Community is working on JSR 282[5], which
contains some fixes and minor extensions of the original specification and will be published as RTSJ 1.1.

Other than realtime, RTSJ also provides helpful standard classes for the general usage in embedded
applications. Its extensions range from asynchronous event handlers—a more convenient way of responding
to events from outside the Java world (referred to as “Happenings” in the RTSJ), such as interrupts and timer events—to specific memory areas outside of the garbage collector’s heap as well as RawMemoryAccess for reading and writing specific memory regions such as memory mapped I/O. With RTSJ, even device drivers can be written completely in Java and can easily be ported to other OSes and to other RTSJ compatible Java Virtual Machines. In order to avoid application delays due to garbage collection even on Java Virtual Machines without a realtime capable garbage collector, RTSJ introduces Immortal Memory and Scoped Memory. Scoped Memory leaves the memory allocation and deallocation to the user, but is much less error-prone than, e.g., malloc is in C.

3 Towards Java Certification—Safety Critical Java

In 2006, a new working group of the Java Community started developing a Java subset for the usage in safety critical applications. This standard, which will be published as JSR 302[8], is being designed to be within the limitations of current DO-178B certification practise. Safety Critical Java (SCJava) aims for Level A, the highest criticality level of DO-178B. In order to achieve this, only a very limited subset of the standard Java classes and the RTSJ will be part of SCJava. One of the restrictions is that no garbage collection is used—which means removing one of the strongest Java features. However, the levels A and B of DO-178B would make certification with a garbage collector too difficult and thus too costly, such that this limitation became necessary. On the other hand, the garbage collector is one of the main reasons for the robustness of Java applications over their counterparts which were written in C, C++, and even Ada.

Automated garbage collection is less error prone and more reliable than manually deallocating memory. The current praxis in safety critical applications, not to allocate any memory at run time, becomes more and more difficult to keep up because of the increasing complexity of such software. This limitation leads to code that is very difficult to understand—which is the opposite of what it meant to achieve. Particularly for safety critical applications, a certified garbage collector would be an important milestone in handling future challenges.

The set of standard classes that comes with SCJava is in the process of being defined. It will roughly be a subset of the Java core classes (java.lang, java.io, java.util, and java.net) and javax.realtime. It will prohibit explicit modification of thread priorities and support the Priority Ceiling Emulation protocol of the RTSJ. In SCJava, an application will consist of a set of AsyncEventHandler; NoHeapRealtimeThread will only be available at level 2. Furthermore, allocation outside of local scoped memory will only be allowed during the initialisation phase.

Despite of its limitations, SCJava is a good first step towards the certification of Java applications according to DO-178B, Level A. Object Orientation, reusability, and platform independence are good reasons to prefer SCJava over Ada or C for the use in safety critical applications.

4 Light at the End of the Tunnel: DO-178C

The biggest problem for the DO-178B certification of Java applications is that this old standard from 1992 does not consider Object Orientated Technology (OOT). While OOT helps software developers in writing reliable and portable applications with much less effort than imperative languages do, it also introduces a number of vulnerabilities. Those need to be identified and addressed in the certification process of safety critical applications. SG-5 of the SC-205/WG-71 plenary is currently working on a new set of objectives, activities, guidance, and guidelines for the certification of OOT software. These will become part of DO-178C[3], the successor to DO-178B. Unlike its predecessors, the new standard finally considers OOT, which substantially eases the certification of Java applications.

DO-178C may provide a more flexible alternative to the rigid structure of DO-178B based on a very simple multitiered development model. The process starts with Tier 0, which take in system design and requirements on the software for a complete aircraft. At each Tier, the input requirements and design are refined to lower level requirements and design. The process continues until an executable “design” emerges from the last Tier (Tier n). The number of refinements n depends on the software development process in use and can vary within a project. Figures 1 and 2 illustrate the old four Tier process of DO-178B and the new multitiered alternative.

OOT introduces a number of features that make software development more efficient and more secure. Before this technology can be used in safety critical projects, analysis is required to make sure that any pitfalls the technology may bring with it can be avoided. The SG-5 Object Oriented Technology working group of the Joint Committee WG-71/SC-205 is currently writing a supplement for DO-178C. The document gives goals, activities, guidance, and guidelines on the certification of Object Oriented applications and is currently in a
Figure 1: The standard certification process from DO-178B ...
draft state. Luckily for Java developers, most vulnerabilities of OOT are alleviated by the safety restrictions of the Java language or by automated memory management.

### 4.1 Inheritance and Redefinition

Inheritance is a core feature of Object Orientation. The encapsulation of code and data provides a safe means for shared code within an application or even reusing it for other applications. Nevertheless, care should be taken to maintain compatibility of subclasses with their parents. In C++, static dispatch and multiple inheritance can make this difficult.

Since a subclass is also a subtype, a general problem with inheritance is maintaining type compatibility. Any subclass should be substitutable for its superclass. Liskov’s Substitution Principle defines type compatibility formally. This can be stated succinctly using preconditions, postconditions, and invariants: compared with those of its superclass, no precondition may be strengthened, and no postcondition or invariant may be weakened in any subclass. Java Annotations are an appropriate way of making such preconditions and postconditions part of the application. The verification activities for any class should also be performed on its subclasses. Vice versa, any class should be verified using its own and its parent’s verification processes.

Another vulnerability is introduced with static dispatch: which implementation of an overwritten method is called, depends on the declared type of the object. This is misleading and confusing: despite a method having been overwritten, the original method might be called on a subclass. The behaviour of C++ is even worse than that: static and dynamic dispatch coexist and can even be mixed within the same class, such that some methods are dispatched statically (which is the default in C++) while others are declared virtual and dispatched dynamically. In Java, method dispatch is always virtual, such that, the method to be called will always be decided based on the real type of an object, not on the declared type.

Multiple inheritance introduces another vulnerability: it is often not clear which implementation of a method is called. Java avoids this confusion by allowing multiple inheritance only for interfaces, but not for implementations. While there is only one possible method implementation to be called, care needs to be taken with contradicting specifications introduced by different interfaces which declare the same method. However, multiple inheritance on the implementation level, as supported by C++, is a significantly complexer problem. In Java, it is sufficient to make sure that all methods are conforming to all of their declarations.
4.2 Parametric Polymorphism

Parametric polymorphism, which is available in Java as Generics or in C++ as Templates, is a type-consistent means to write reusable code whenever sub-typing is not possible or convenient. An example is a List, each of which is should only have a certain type of element, but should be usable for all element types.

Parametric polymorphism is a strong feature of Object Oriented languages. For certification, each unique instantiation of a parametrised type or combination of types needs to be verified. Generics in Java are type safe, and consistency is checked at compile time as far as possible.

4.3 Type Conversion

OO Type conversion, as well as in imperative programming languages, is sometimes necessary, but can cause unexpected behaviour. The strong type system of Java tries to detect type inconsistencies at compile time as far as possible, and throws an exception at runtime in all other cases. While throwing an exception is preferable over working with corrupted data (like C or C++ would do), certification requires a static proof of correctness. Static type checking, e.g., based on Data Flow Analysis, can help to ensure the correct typing of an application. Theoretically, Data Flow Analysis can also be used for analysing C or C++ programs, but the result is less reliable than in Java on any language that does not prohibit pointer arithmetics.

4.4 Overloading

When used with care, overloading can improve readability and code maintenance. Along with implicit type conversion, which is common in Object Oriented programming languages, Overloading may cause ambiguity. In order to avoid this, guidelines should address the cases in which overloading is allowed and discourage the use of implicit type conversion.

4.5 Exception Management

Most OO languages support throwing and handling exceptions instead of returning from a method with an ordinary return value. While Exceptions are a safe and convenient way to deal with exceptional situations, they introduce another vulnerability: exceptions might leave the application in an inconsistent state in case an unexpected exception is handled very low in the call stack or even not handled at all.

Java supports both checked and unchecked exceptions. Unless a method declares throwing a checked exception, it must be handled whenever this exception might be thrown within the scope of this method. The need to handle checked exceptions explicitly makes their usage safe. Unchecked exceptions are used for runtime errors such as division by zero, bounds checks, and range checks, and usually mainly thrown implicitly by the VM in situations, which would cause the system to either crash or run in an inconsistent state if this situation had occurred in a programming language without exception management.

aicas is working on a Data Flow Analysis based tool which can prove that all unchecked exceptions are handled by the application and detect all occurrences of unhandled unchecked exceptions. This approach ensures highest safety for critical applications.

4.6 Dynamic Memory Management

Complex tasks often require temporary data. There are several possibilities to deal with the allocation and deallocation of this data, the safest and most convenient of which is an automated garbage collector. SG-5 has determined the following vulnerabilities related to dynamic memory management. In Java, all but Heap memory exhaustion can be avoided by a realtime capable automated garbage collector.

a.) Ambiguous References: Objects overlap in the same memory region. This can be avoided by an allocator which ensures exclusivity, provided that pointer arithmetics is not possible (as it is in Java) or not used (which would be necessary, but hard to ensure, e.g., in C or C++). Ambiguous references can also be a consequence of the deallocation of an object that was still in use.

b.) Fragmentation Starvation: A new object cannot be allocated due to memory fragmentation. To prevent this, memory should be organised in a way that ensures all allocations will succeed, provided that the system has enough of free memory. Many Java garbage collectors can ensure this. C applications, especially in Embedded Systems, are often vulnerable for fragmentation starvation.
c.) **Deallocation Starvation**: Garbage objects may not be freed, or at least not fast enough, causing the application to run out of memory. Some Java garbage collectors can guarantee that they reclaim memory fast enough to prevent this.

d.) **Heap Memory Exhaustion**: The Heap may be insufficient for the application to run. The application needs to ensure that all simultaneously live objects fit into the available memory. Again, data flow analysis is a safe approach to ensure this.

e.) **Premature Deallocation**: Objects may be removed although they are still in use. This causes dangling pointers, which is a common problem in C or C++ applications. However, all Java garbage collectors should be able to avoid this vulnerability.

f.) **Lost Update and Stale Reference**: In order to prevent fragmentation starvation, some systems move objects to a different location. Those need to make sure that modifications of an object which is being moved also affect the new location. Similarly, any read access to this object should return the new data of the modified object. Non-moving garbage collectors or one that move objects atomically are safe to use here.

g.) **Indeterministic Allocation or Deallocation**: Dynamic memory management could interrupt the application unexpectedly. Most automated garbage collectors that prevent the other vulnerabilities in this section, cannot guarantee determinism. For example, the Boehm garbage collector, which is commonly used in C++ applications, is vulnerable in this respect. The garbage collector of the JamaicaVM Java Virtual Machine guarantees determinism and avoids all other vulnerabilities in this section, with the exception of Heap memory exhaustion.

Various techniques exist to deal with those vulnerabilities: manual heap allocation, which is commonly used in Embedded C applications that do not need to be certified, leave most of them to be dealt with by the application developers. Object pooling is only slightly better: rather than allocating and initialising a complex object from scratch when it is needed and destroying it again afterwards, the object could be taken from a prefilled object pool, which is much faster than the allocation. When the object is no longer needed, it is sent back to the object pool rather than destroyed. While object pools avoid the typical fragmentation of a malloc, they introduce a new kind of fragmentation: while free objects might still exist in some of the object pools, the pool holding the kind of objects to be allocated might be empty. Unlike manual heap allocation, this is at least some kind of fragmentation which the application developers have a chance for dealing with manually. An advantage of object pooling is the fast allocation time as long as there are enough objects left in the pool.

Stack allocation is used to store local object on the call stack, removing them automatically on method exit. While this lowers the danger of fragmentation, it also limits the extent to which frames can be shared between threads and is only usable for local objects. Scope based object management is slightly more flexible, because it allows for several threads to enter a certain frame simultaneously. At the same time, scope allocation causes a higher risk of fragmentation. The referencing rules of scoped memory prevent dangling references, which is not true for stack allocation in languages such as C and C++.

Automated garbage collection is the most convenient and, if it solves the vulnerabilities identified by the SG-5, safest way to deal with dynamic memory. Care should be taken with the selection of the garbage collector: some cannot ensure that free memory can be detected early enough to avoid deallocation and fragmentation starvation or that an upper bound of allocation and deallocation time is not exceeded. The realtime Garbage Collector of JamaicaVM is able to guarantee all of this.
The following table summarises to what extent the various techniques deal with the vulnerabilities of dynamic memory management.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Ambiguous References</th>
<th>Fragmentation Starvation</th>
<th>Deallocate Starvation</th>
<th>Virtualisation Starvation</th>
<th>Heap Memory Exhaustion</th>
<th>Premature Deallocate</th>
<th>Lost Update and Stable Reference</th>
<th>Indeterministic Allocation or Deallocate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Heap Allocation</td>
<td>X</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>✓</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Object Pooling</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>N/A</td>
<td>✓</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Stack Allocation</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
<td>✓</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Scope Allocation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>N/A</td>
<td>✓</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Automated Heap Allocation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

= prevented automatically, ✓ by the application, N/A = not applicable, □ = difficult to ensure

4.7 Virtualisation

Virtualisation generally improves the portability and reusability of application code and typically reduces the complexity of applications. Java applications generally run in such a virtual environment. The main vulnerability here is that interpreted code is not sufficiently validated, because it is treated as data rather than code. DO-178C generally allows for virtualisation, but requires that each layer is verified independently, i.e., when certifying the interpreter, its input can be treated as data, but an additional verification of the interpreted code, in which the interpreter is treated as execution platform is also required.

5 Summary

Object Oriented Technology in general and Java in particular improve the efficiency of developers by reducing the complexity of large applications. Automated Garbage Collection, a strong type system, and the prohibition of pointer arithmetics and multiple inheritance reliably prevent many hard-to-detect errors which are typical in C and C++ applications.

RTSJ is a good, safe, and wide-spread extension for the usage of Java in Embedded Systems and realtime applications. While it is also a good fundament for safety critical applications, RTSJ alone is not sufficient for a certification, e.g., according to DO-178B. The combination of RTSJ with a realtime Garbage Collector is a good match for certifications up to level C.

With Safety Critical Java, a Java subset is being developed which even addresses the highest Level A of the DO-178B certification standard. Unfortunately, the strict rules of DO-178B just prohibit the usage of the automated Garbage Collector, one of Java’s most effective safety features. The successor of the standard, DO-178C, which is currently being developed, finally considers Object Orientation and automated Garbage Collection. This will be a mile stone in the certification of the safety critical applications of the future, which have to deal with much more complex tasks than the applications currently in use and therefore require modern development tools.

References


