

# Draft Guidelines for Scalable Java<sup>1</sup> Development of Real-Time Systems

Kelvin Nilsen, Ph.D., Chief Technology Officer  
Aonix

Copyright © 2004, 2005

Aonix North America  
5040 Shoreham Place  
Suite 100  
San Diego, CA 92122  
USA

Aonix  
Batiment B  
66/68, Avenue Pierre Brossolette  
92247 Malakoff cedex  
France

Tel: 1-800-972-6649  
or 1-858-457-2700  
Fax: 1-858-824-0212

Tel: +33 1 4148-1000  
Fax: +33 1 4148-1020

May 6, 2005 8:45 am

---

1. Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

# Table of Contents

Draft Guidelines for Scalable Java Development of Real-Time Systems	1
Section 1. Motivation	1
Figure 1: Tabulated Survey Responses	2
Section 2. Terminology and Guiding Principles	3
Figure 2: Decision Tree for Selecting Between Alternative Java Technologies	5
Section 3. Soft Real-Time Development Guidelines	6
Section 4. Hard Real-Time Development Guidelines	9
Section 5. Safety-Critical Development Guidelines	14
Section 6. Enforcement of Style Guidelines	17
Appendix A: Hard Real-Time Subset of the J2SE API	38
Section 1. Subset of the JDK 1.5 <code>java.lang</code> Package	38
Section 2. Subset of the JDK 1.5 <code>java.lang.annotation</code> Package	61
Section 3. Subset of the JDK 1.5 <code>java.lang.reflect</code> Package	62
Section 4. Subset of the JDK 1.5 <code>java.io</code> Package	65
Section 5. Subset of the JDK 1.5 <code>java.util</code> Package	65
Appendix B: Hard Real-Time Subset of Real-Time Specification for Java	66
Appendix C: Hard Real-Time Extensions to Real-Time Specification for Java	79
Section 1. Annotations to Support Consistent Initialization of Class Variables	79
Figure 3: Dependency Graph for Topological Sort	83
Section 2. Starting up a Safety-Critical Java Application	85
Section 3. Sharing Objects with Traditional Java Domain	85
Figure 4: Annotated Source code for <code>Thermostat</code> Class	86
Figure 6: Hard Real-Time View of <code>Thermostat</code> Class	89
Figure 5: Compilation and Verification of Interface Components	89
Figure 7: Traditional Java View of <code>Thermostat</code> Class	90
Section 4. Annotation to Support Static Analysis of Hard Real-Time Components	93
Section 5. Annotations to Support Maintenance of Multi-Linked Data Structures	100
Section 6. Annotations to Support Static Enforcement of Referential Integrity	104
Section 7. Using Nested Scopes for Composition of Modular Software Systems	108
Figure 8: Criticality and Timeliness Requirements for Architectural Components	110

# Table of Contents

Figure 9: Inherent Dependency Analysis Between Components	111
Figure 10: Dependency Analysis of Modules and Interfaces	112
Figure 11: Organization of Safety-Critical Memory Partition	113
Figure 12: Organization of Hard Real-Time Mission-Critical Partition	114
Figure 13: Code to Initialize the <i>Level0</i> Class	115
Figure 14: Code to Initialize the <i>Level1</i> Class	116
Figure 15: Variable Declarations and Constructor for <i>Level2</i> Class	117
Figure 16: The <i>run()</i> Method for <i>Level2</i> Class	118
Figure 17: The <i>doMaintenance()</i> method for <i>Level2</i> Class	119
Figure 18: <i>Level2</i> Methods to Hot-Swap Certain Modules	120
Section 8. Standard Hard Real-Time Extended API for Safety-Critical Java	121
Section 9. Standard Utility Libraries for Hard Real-Time Development	208
Section 10. Standard Hard Real-Time Extended API for Mission-Critical Java	212
Appendix D: Index of Program Library Components	223

# Draft Guidelines for Scalable Java Development of Real-Time Systems

*Kelvin Nilsen, Ph.D., CTO, Aonix North America*

*This document establishes programming guidelines to assure that real-time Java software satisfies reliability requirements and is economically maintainable, portable, and scalable. The guidelines are based on making effective use of the traditional J2SE Java in combination with appropriate profiles of the Real-Time Specification for Java.*

## 1. Motivation

As one of the driving forces behind the standardization of safety-critical and mission-critical Java technologies within the Open Group's Real-Time and Embedded Forum, Aonix conducted a survey of various prospective users of the technology. As of the date of this draft document, we have collected nine survey responses. Four responses came from Europe and five from the United States. Four respondents represented companies who develop safety-critical components for commercial and defense avionics. One respondent represented the satellite software division of a well-known telecommunications company. Two were expert consultants in the application of rigorous software process to support certification of safety-critical systems. One represented a government aerospace agency. And the last one was a university professor with internationally recognized expertise in static analysis of software. Survey responses are tabulated in Figure 1.

While there was not 100% agreement, the surveys did reveal interesting clusters of common sentiment. Our interpretation of the responses follows:

1. Most respondents were somewhat comfortable using the RTSJ for soft real-time, but most were not comfortable using RTSJ for hard real-time development.
2. There was strong agreement that large software development projects need to define an RTSJ "profile" to be targeted as the common platform for the development team.
3. Though some respondents believe their organizations are capable of defining their own RTSJ profiles, strong preference was expressed towards defining common profiles in open, consensus-based standards organizations.
4. There was strong support expressed for the notion of families of complementary profiles that allow synergy between members of the family, such as profiles for both safety-critical and mission-critical development which share many libraries and development tools.
5. There was also very strong backing for the notion that organizations need to enforce that programmers adhere to the development guidelines that comprise a particular RTSJ profile.
6. Most respondents believe that in order to achieve their objectives, profiles must both subset and superset the capabilities of the RTSJ standard.
7. People need more information before deciding on the best "standard" for safety-critical Java.

The developer guidelines published in this document were created in response to a request from one of the survey participants. This participant requested that we help them establish guidelines that could be used

**(Permission granted to reproduce and distribute as a complete document, without modification)**

Not Qualified to Judge	Strongly Disagree	Somewhat Disagree	Undecided	Weakly Agree	Strongly Agree	
11%	0%	11%	33%	11%	33%	The full RTSJ (not a specific profile thereof) is a good platform for developing large, complex, dynamic soft real-time systems.
11%	56%	11%	22%	0%	0%	The full RTSJ (not a specific profile thereof) is a good platform for developing small, resource constrained, high-performance hard real-time systems.
0%	11%	0%	22%	11%	56%	Before undertaking a large-scale real-time development effort using the RTSJ (whether hard or soft real-time, or both), a profile of the RTSJ should be defined to restrict and/or enhance the capabilities of the RTSJ.
0%	38%	0%	38%	25%	0%	My organization is comfortable defining its own RTSJ profiles for its own projects.
0%	11%	0%	0%	33%	56%	RTSJ profiles should be standardized through an open, consensus-based standards organization.
0%	0%	22%	11%	11%	56%	It is very important that a family of complementary RTSJ profiles provide compatibility between profiles within the family.
0%	0%	0%	22%	11%	67%	When developing and maintaining code for a large-scale RTSJ effort that targets a particular RTSJ profile, it is important to enforce compliance with that profile.
0%	22%	33%	22%	22%	0%	An RTSJ profile should only subset from the minimal required features of the RTSJ specification, without imposing particular interpretations or requiring capabilities that are not present in every compliant RTSJ implementation.
0%	11%	11%	11%	22%	44%	An RTSJ profile may clarify vague or ambiguous capabilities of the RTSJ.
0%	11%	22%	11%	22%	33%	An RTSJ profile may require the existence of run-time libraries that are not necessarily present in every compliant RTSJ implementation.
0%	0%	22%	11%	22%	44%	An RTSJ profile may require the existence of special development tools and/or implementation techniques that are not necessarily available in every compliant RTSJ implementation.
0%	11%	22%	0%	22%	44%	In order to effectively meet the needs of safety-critical development, it is essential that the safety-critical profile specify certain libraries, development tools, and implementation techniques that are not necessarily present in every compliant RTSJ implementation.
44%	0%	0%	44%	11%	0%	The draft safety-critical semantics description and accompanying API is close to what is needed for safety-critical development.

Figure 1: Tabulated Survey Responses

within his organization today, even before the Open Group standards have become official. Their desire was to maintain compatibility with existing standards, such as J2SE and RTSJ, but also work towards compatibility with the anticipated future standards for safety-critical and hard real-time mission-critical technologies. Given the need to steer current real-time Java development efforts within their organization, this organization felt that the “working draft” of the Open Group standards process was as good a starting point as any.

Our sense is that the only way for the various Open Group participants to gain confidence in their assessment of particular tool or syntactic features for the safety-critical Java standard is to apply these technologies experimentally to realistic representative problems. With this in intent, we are putting forth these developer guidelines and are proceeding to implement the related software development tools and library components even before the standards have been established. We encourage all companies who have vested interests in establishing strong standards for safety-critical and mission-critical Java to experiment with these early stage technologies.

## 2. Terminology and Guiding Principles

As a very high-level programming language, Java offers programmer and software maintenance productivity benefits that range from two to ten-fold over uses of C and C++. By carefully applying Java technologies to embedded real-time systems, software engineers are able to deliver higher software quality, increased functionality, and greater architectural flexibility in software systems. It is essential that programmers recognize the strengths and weaknesses of particular Java technologies in order to judge the relevance of each Java technology to each set of software requirements.

**What is Real Time?** Within this document, we distinguish between two different kinds of real-time constraints:

### Hard Real Time

Describes systems in which an action performed at the wrong time has zero or possibly negative value. The connotation of “hard real time” is that compliance with all timing constraints is proven using theoretical static analysis techniques prior to deployment.

### Soft Real Time

Describes systems in which an action performed at the wrong time (either too early or too late) has some positive value even though it would have had greater value if performed at the proper time. The expectation is that soft real-time systems use empirical (statistical) measurements and heuristic enforcement of resource budgets to improve the likelihood that software complies with timing constraints.

Note that the difference between hard real-time and soft real-time does not depend on the time ranges specified for deadlines or periodic tasks. A soft real-time system might have a deadline of 100  $\mu$ s, while a hard real-time system may have a deadline of 3 seconds.

Modern operating systems and CPU architectures strongly favor the use of empirical rather than analytical enforcement of timing constraints. It is prohibitively expensive to analyze cache misses, pipeline stalls, out-of-order execution and branch prediction for all but the simplest code on the most primitive of processors. Thus, there is strong incentive to migrate as much functionality as possible into the soft real-time domain. The typical approach consists of the following:

**(Permission granted to reproduce and distribute as a complete document, without modification)**

- Use dedicated hardware with abundant buffering capabilities to implement hard real-time functionality.
- Design and implement protocols that are robust in the face of occasional deadline misses. For example, if data is lost due to a buffer overrun, provide mechanisms to request retransmission of the data or introduce an algorithm that automatically derives approximate values to replace any lost data.

**Safety-Critical Development.** Within this document, we characterize safety-critical software as software that must be certified according to DO-178B or equivalent guidelines. Certification guidelines impose strict limits on software practices, including peer review, traceability analysis, and software testing.

Throughout the remainder of this document, we use the term safety-critical to specifically represent DO-178B levels A and B, unless indicated to the contrary. Managers who are responsible for development of level C components must carefully weigh the benefits of using higher level programming language features (e.g. soft real-time Java or hard real-time mission-critical Java) to reduce development costs versus the benefits of using the safety-critical Java profile to reduce the certification costs.

**Automatic Garbage Collection.** One of the key reasons why Java developers are more productive than C and C++ developers is because of automatic garbage collection. According to a study performed by Xerox Palo Alto Research Center in the early 1980s, automatic garbage collection reduces programming effort associated with large, complex software systems by approximately 40%. These benefits are amplified significantly in the Java environment because automatic garbage collection is the foundation upon which millions of lines of COTS (commercial off-the-shelf) software including all of the standard Java libraries are based. If you remove garbage collection from the Java environment, not only do you make it more difficult to develop new software, but you also preclude the use of all existing Java library code.

The power of garbage collection comes with a cost. Traditional Java implementations occasionally pause execution of Java threads to scan all of memory in search of objects that are no longer being used. These pauses can last tens of seconds with large memory heaps. Memory heaps ranging from 100 Mbytes to a full Gigabyte are being used in certain mission-critical systems. The 30-second garbage collection pause times experienced with traditional Java virtual machines are incompatible with the real-time execution requirements of most mission-critical systems.

Special real-time virtual machines have been implemented to support preemptible and incremental operation of the garbage collector. With these virtual machines, the interference by garbage collection on application code can be statistically bounded, making this approach suitable for soft real-time systems with timing constraints measured in the hundreds of microseconds.

One of the costs of automatic garbage collection is the overhead of implementing sharing protocols between application threads. Application threads are continually modifying the way objects relate to each other within memory while garbage collection threads are continually trying to identify objects that are no longer reached from any threads in the system. This coordination overhead is one of the main reasons that compiled Java programs run at  $\frac{1}{3}$  to  $\frac{1}{2}$  the speed of optimized C code.

The complexity of the garbage collection process and of any software that depends on garbage collection for reliable execution is beyond the reach of cost-effective static analysis to guarantee compliance with all hard real-time constraints. Thus, we do not recommend the use of automatic garbage collection for software that has hard real-time constraints.

**Choosing the Right Tool for the Job.** The embedded real-time market has been described as a thousand different niches. We recognize that each critical software component represents different requirements and economic tradeoffs. We recommend that managers consider the decision tree in Figure 2 to determine how

to implement particular capabilities. Note that the soft real-time Java development guidelines are generally

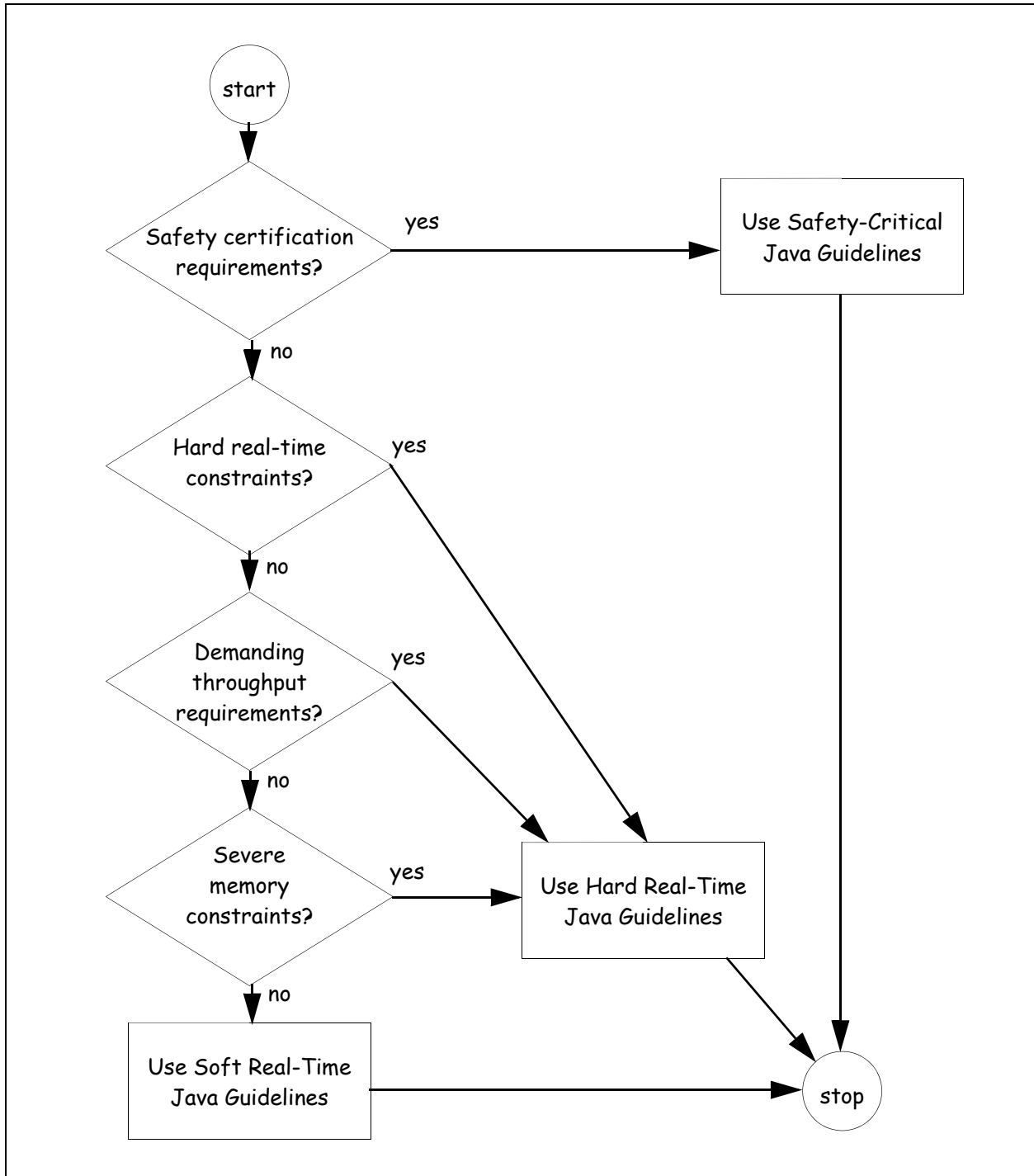


Figure 2: Decision Tree for Selecting Between Alternative Java Technologies

preferred unless there is a specific constraint that precludes them. This is because soft real-time Java offers the highest developer and software maintenance productivity.

(Permission granted to reproduce and distribute as a complete document, without modification)

### 3. Soft Real-Time Development Guidelines

---

#### **Rule 3.1: Use the Java 2 Standard Edition (J2SE) Platform**

---

The benefits that Java brings to soft real-time mission-critical systems are most relevant to large, complex, dynamic applications. Since the J2ME platform represents an incompatible subset of full J2SE, it does not provide access to J2SE-standard COTS library components. If applications require J2EE capabilities, obtain the specific J2EE libraries that are required and run them on a soft real-time J2SE platform. Alternatively, run the required J2EE functionality on traditional (non real-time) JVM platforms which communicate with the soft real-time JVM machines using RMI or other networking protocols.

---

#### **Rule 3.2: Baseline a Particular Version of the J2SE Libraries**

---

For any given development project, it is necessary to standardize on a particular version of the J2SE libraries (1.2, 1.3, 1.4, 1.5?). Document this decision to all developers and managers.

---

#### **Rule 3.3: Follow “Best Practices” Recommendations for Java Development**

---

The software quality measures are very similar for soft real-time Java and traditional Java. Unless specified to the contrary in this section, follow the accepted guidelines for writing portable and maintainable J2SE code.

---

#### **Rule 3.4: Use JFace and SWT for Graphical User Interfaces**

---

Most mission-critical software does not require graphical user interfaces. If soft real-time systems do require graphical user interfaces, use the open-source **SWT** and **JFace** libraries instead of the proprietary **AWT** and **Swing** components. **SWT** and **JFace** run in less memory and run faster than **SWT** and **Swing**.

---

#### **Rule 3.5: Use Cooperating Hard Real-Time Components to Interface with Native Code**

---

The JNI protocol introduces significant data marshalling overhead when objects are shared between the Java and native environment. Furthermore, the sharing protocols may expose Java objects and “private” virtual machine data structures to undisciplined C components, introducing the risk that misbehaving C code will compromise the integrity of the virtual machine environment. Experience of existing customers in several real projects involving hundreds of man years of development document that these risks are real, having cost development teams significant effort and calendar time to correct errors introduced into the Java environment by C developers writing JNI components.

Better performance and stronger separation of concerns is realized by implementing all interfaces to native code as cooperating hard real-time components.

---

**Rule 3.6: Use Cooperating Hard Real-Time Components to Implement Performance-Critical Code**

---

If the throughput of certain soft real-time components is not sufficient to meet performance requirements, implement the required functionality as cooperating hard real-time components. Because the code generation model for hard real-time components does not need to coordinate with garbage collection, these components generally run two to three times faster than soft real-time Java components.

---

**Rule 3.7: Use Cooperating Hard Real-Time Components to Interact Directly with Hardware Devices**

---

If the soft real-time component needs to communicate directly with hardware devices which are not represented by operating system device drivers, implement the device driver as a cooperating hard real-time component. If the operating system provides a device driver that represents this device as a file, use the standard `java.io` or `java.nio` libraries to access the device. If the operating system provides a device driver with a different API than the file system, use a cooperating hard real-time component to implement the interface to the device driver.

---

**Rule 3.8: Restrict the Use of “Advanced Libraries”**

---

Certain standard Java libraries are not available in certain embedded environments because the underlying operating system or hardware is missing desired capabilities. Among the libraries that may not be available on all platforms, listed in decreasing order of portability concern, are:

- `JFace` and `SWT` libraries: These graphical libraries are only available on systems that have graphical hardware and the `SWT` integration software required to drive the graphical hardware.
- `java.nio` libraries: Many embedded operating systems do not support asynchronous I/O.
- `java.io` libraries: Some embedded targets have no notion of `stdin`, `stdout`, or `stderr`. Some embedded targets have no notion of non-volatile file storage.
- `java.net` libraries: Some embedded targets have no network connectivity.

Recognize that the use of these libraries may limit the portability of code and may contribute to the future maintenance burden.

---

**Rule 3.9: Isolate JVM Dependencies**

---

Existing soft real-time virtual machines differ in how they support certain important mission-critical capabilities. Wrap all JVM dependencies in special classes that can be given extra attention if the code must be ported to a different JVM. Specific services that require this handling include:

- High-precision timing services: obtaining real-time with greater precision than 1 ms; drift-free `sleep()`, `wait()`, and `join()` services.
- CPU-time accounting: How much CPU time consumed by each thread? How much CPU time consumed at each priority level?

**(Permission granted to reproduce and distribute as a complete document, without modification)**

- Garbage collection pacing: How to monitor the memory allocation behavior of the application software and the effectiveness of GC? How to schedule GC to maintain pace with allocation rates?
- Scheduling: If a virtual machine offers high-level scheduling support, such as earliest-deadline first or maximum accrued utility scheduling, the scheduling and synchronization services should be isolated within a centralized API.

---

**Rule 3.10: Carefully Select an Appropriate Soft Real-Time Virtual Machine**

---

One of the most important decisions in determining the success of a soft real-time Java development effort is selection of a suitable JVM. Each development project has unique requirements and constraints, so it may be necessary to independently evaluate the relevance of various available virtual machine products for each development effort. In selecting a virtual machine, consider at minimum each of the following issues:

- ❑ Real-Time garbage collection should have a maximum preemption latency and should be incremental so that when the garbage collector is preempted by higher priority application threads, it can resume with the next increment of work when the application thread relinquishes the CPU. The garbage collector should defragment the heap in order to assure reliable long-running operation or should provide some alternative mechanism to avoid reliability problems resulting from fragmentation of the heap. And it must accurately reclaim all dead memory rather than reclaiming only a conservative approximation of the dead memory. Finally, it must be paced to assure that memory is reclaimed at rates consistent with the application's steady-state demand for new memory allocation.
- ❑ All synchronization locks must implement priority inheritance. All wait queues must be ordered according to thread priorities.
- ❑ The virtual machine needs to provide monitoring facilities to allow supervisory threads to observe and measure the real-time resource requirements of individual components. Among required capabilities are the ability to determine how much CPU time is consumed by particular threads, how much CPU time is consumed by the garbage collection thread(s), the rates at which particular threads are allocating memory, and the total amount of memory being retained as live.
- ❑ Determine which release level of the J2SE libraries are required for a particular project (1.2, 1.3, 1.4, 1.5?) and assure that the vendor is able to support the desired library version throughout the duration of your development project.
- ❑ Assure that the virtual machine provides libraries for high-precision time measurements, and for drift-free `wait()`, `join()`, and `sleep()` services.
- ❑ If the system is statically compiled and loaded, assure that the virtual machine is supported by appropriate Ahead-of-Time compilation and linking tools.
- ❑ If the system must dynamically load components, assure that the dynamic class loader can be configured to run at lower priority than the ongoing real-time application workload. If the dynamic class loader must perform JIT compilation, assure that the JIT compiler can be configured to support eager linking and translation, meaning that all components are fully resolved and translated when the first of the interdependent modules is loaded, rather than deferring JIT translation until

**(Permission granted to reproduce and distribute as a complete document, without modification)**

the moment each code module is first executed. Some systems need to dynamically load components which have themselves been ahead-of-time compiled. Verify this capability is supported if relevant to your project requirements.

- ❑ Assure that the virtual machine includes necessary development tools, including symbolic debugging of both interpreted and compiled code and run-time performance and memory usage profiling.
- ❑ If the planned development project may require integration with cooperating hard real-time components, assure that the virtual machine includes support for cooperating hard real-time Java components.

#### 4. Hard Real-Time Development Guidelines

The recommendations of this section are based on standards for safety-critical and mission-critical Java which are being developed within the Open Group.

---

##### **Rule 4.1: Use a Hard Real-Time Subset of the Standard Java Libraries**

---

There is no automatic garbage collection in the hard real-time domain so many of the standard Java libraries will not function reliably. Other motivations to restrict usage of the standard libraries are (1) to reduce the standard memory footprint and (2) to reduce the amount of code that must be certified in case safety certification requirements must be satisfied. The set of libraries approved for use in hard real-time Java applications, which only supports `NoHeapRealtimeThread` and forbids the use of `java.lang.Thread` and `RealtimeThread`, is detailed in Appendix A.

---

##### **Rule 4.2: Use a Hard Real-Time Subset of the Real-Time Specification for Java**

---

The full RTSJ includes many capabilities that are not portable between different compliant implementations. Furthermore, supporting the full generality of the RTSJ imposes certain performance-limiting restrictions on the implementation. The set of RTSJ libraries that are approved for use in hard real-time Java applications is detailed in Appendix B.

---

##### **Rule 4.3: Use Enhanced Replacements for Certain RTSJ Libraries**

---

Certain RTSJ libraries lack the features desired for hard real-time and safety-critical development. Analogous replacement libraries are available in the `javax.realtime.util.sc` package. Use these replacement libraries instead of the traditional RTSJ libraries. The specific replacement libraries, which differ only slightly from their RTSJ counterparts, are listed below:

- `AbsoluteTime`
- `AperiodicParameters`
- `AsyncEvent`
- `BoundAsyncEventHandler`
- `Clock`

**(Permission granted to reproduce and distribute as a complete document, without modification)**

- `HighResolutionTime`
- `NoHeapRealtimeThread`
- `OneShotTimer`
- `PeriodicParameters`
- `PeriodicTimer`
- `RelativeTime`
- `ReleaseParameters`
- `SizeEstimator`
- `SporadicParameters`
- `Timer`

---

**Rule 4.4: Use Standard Extended Libraries for I/O Ports and Interrupt Handling**

---

When implementing low-level access to I/O devices and interrupt handling, use the standard RTSJ library extensions described in Appendix C.

---

**Rule 4.5: Assure Availability of Supplemental Libraries**

---

Appendix C describes standard libraries that are assumed to exist in all compliant hard real-time execution environments. If particular applications require additional libraries beyond this minimal set, assure that the libraries are available for all intended target platforms.

---

**Rule 4.6: Use An Intelligent Linker and Annotations to Guide Initialization of Static Variables**

---

In traditional Java, class variables are to be initialized “immediately before first use”. This requires run-time checks, introduces non-determinism into the worst-case execution-time analysis, and hinders efficient translation of programs for native execution. Further, it introduces certain race conditions in which the initial values of particular class variables (even the values of certain `final` variables) depend on the sequence in which classes are accessed (and initialized). Use an intelligent static linker guided by `@StaticDependency` and `@InitializeAtStartup` annotations to perform initialization of all static variables, as described in Appendix C.

---

**Rule 4.7: Use Only 128 Priority Levels for `NoHeapRealtimeThread`**

---

The official RTSJ specification states that a compliant implementation must provide at least 28 priorities, but may support many more. For hard real-time mission-critical development, application software should limit its use of priorities to the range from 1 though 128. Vendors can readily support this priority range as a standard hard real-time mission-critical platform.

(Permission granted to reproduce and distribute as a complete document, without modification)

---

**Rule 4.8: Do Not Instantiate `java.lang.Thread` or `javax.realtime.RealtimeThread`**

---

The only threads allowed to run in a hard real-time program are instances of `NoHeapRealtimeThread`.

---

**Rule 4.9: Use `NATIVE`-mode `NoHeapRealtimeThread` for Invocation of Native Methods**

---

Only threads that are instantiated with the `NATIVE` mode parameter are allowed to invoke native methods. Note that a transfigured `NATIVE`-mode thread is also not eligible for invocation of native methods.

---

**Rule 4.10: Do Not invoke any Java synchronization or blocking services from within `NATIVE`-mode `NoHeapRealtimeThread`**

---

Because different real-time operating systems may schedule native threads differently, it is not possible to guarantee that native-scheduled threads will properly honor the priority inversion avoidance protocols for Java synchronization.

---

**Rule 4.11: Invoke `javax.realtime.util.sc.NoHeapRealtimeThread.transfigure()` from `NATIVE`-mode thread before executing Java synchronization**

---

Invoking `transfigure()` causes the current `NATIVE`-mode thread to be scheduled by the real-time Java scheduler. As a transfigured thread, it is now legal to execute Java synchronization and blocking code.

---

**Rule 4.12: Invoke `javax.realtime.util.sc.NoHeapRealtimeThread.condescend()` from transfigured `NATIVE`-mode thread in order to invoke native methods**

---

Invoking `condescend()` from a transfigured `NATIVE`-mode thread causes the thread to be scheduled once again by the underlying operating system. As such, the thread is eligible to invoke native methods but may not execute Java synchronization until it once again invokes `transfigure()`.

---

**Rule 4.13: Preallocate `Throwable` Instances**

---

The traditional Java convention of allocating a new `Throwable` each time an exceptional condition is encountered is not compatible with limited-memory hard real-time development practices. Preallocate necessary `Throwable` objects in scopes that are sufficiently visible that they can be seen by the intended `catch` statement. Throw the preallocated `ImmortalMemory Throwable` instances available in `javax.realtime.util.sc.PreallocatedExceptions` and `javax.realtime.util.mc.PreallocatedExceptions` when appropriate.

---

**Rule 4.14: Restrict Access to Throwable Attributes**

---

In a traditional Java environment, memory is allocated to represent private information associated with each thrown `Throwable`. Because a hard real-time environment is assumed to have limited memory resources and no automatic garbage collection, the typical hard real-time programming style avoids allocation of memory for each thrown exception. One fixed-size buffer holding up to 20 `StackTraceElement` objects is maintained for each thread. Each time an exception is thrown, this buffer is overwritten with no more than 20 of the inner-most nested method activation frames. The buffer's contents can be copied by invoking `Throwable.getStackTrace()` before any other `throw` statements are executed by the thread. Any code that attempts to access more than 20 stack frames, or delays invocation of `Throwable.getStackTrace()` until after a second `Throwable` has been thrown, will not run reliably in the hard real-time environment.

---

**Rule 4.15: Annotate All Program Components to Indicate Scoped Memory Behaviors**

---

In order to enable static analysis to prove referential integrity without the need for run-time fetch and store checks, programmers must annotate their software to identify variables that might hold references to objects allocated in temporary memory scopes. The annotation guidelines are described in Appendix C.

---

**Rule 4.16: Carefully Restrict Use of Methods Declared with `@AllowCheckedScopedLinks` Annotation**

---

Methods with this annotation may terminate with a run-time exception resulting from inappropriate assignment operations. Automated static analysis tools are not able to guarantee the absence of these run-time exceptions, and reference assignment operations contained within these methods will run slower than other code because each assignment must be accompanied by a run-time check. For each method that is declared with the `@AllowCheckedScopedLinks` annotation, programmers should provide commentary explaining why they believe the code will not violate scoped-memory referential integrity rules.

---

**Rule 4.17: Carefully Restrict Use of Methods Declared with `@ImmortalAllocation` Annotation**

---

As a rule of thumb, `ImmortalMemory` should only be allocated during application startup. Any other allocation of `ImmortalMemory` introduces the risk that the supply of `ImmortalMemory` will become exhausted in a long-running application.

---

**Rule 4.18: Use `@StaticAnalyzable` Annotation to Identify Methods with Bounded Resource Needs**

---

The `@StaticAnalyzable` annotation identifies methods that have bounded CPU time and memory needs. For any program component declared with the `@StaticAnalyzable` annotation, programmers should provide `StaticLimit` assertions to identify iteration limits on loops and other resource constraints. Complete details are provided in Appendix C.

(Permission granted to reproduce and distribute as a complete document, without modification)

---

**Rule 4.19: Use Hierarchical Organization of Memory to Support Software Modules**

---

Organize software modules to support modular composition of components so that all memory allocation for individual components, including the memory for all of the threads that comprise the software module, is hierarchically organized. The memory for the complete module is incrementally divided into memory for sub-modules. Each sub-module may further divide its memory for smaller sub-modules. All memory reclamation is handled in last-in-first-out order with respect to allocation sequence. Use the `ThreadStack` data abstractions described in Appendix C.

---

**Rule 4.20: Use the `@TraditionalJavaShared` Conventions to Share Objects with Traditional Java**

---

When it is necessary or desirable to share hard real-time data and/or control abstractions with the traditional Java domain, use the `@TraditionalJavaShared` and `@TraditionalJavaMethod` annotations and the protocols described in Appendix C to arrange the sharing of selected objects.

---

**Rule 4.21: Avoid `synchronized` Statements**

---

When synchronization is required, use `synchronized` methods instead of individual statements.

---

**Rule 4.22: Restrict invocations of `Object.wait()`, `Object.notify()`, and `Object.notifyAll()` to `synchronized` methods that correspond to the target of the `wait()`, `notify()`, or `notifyAll()` invocation**

---

To facilitate readability, maintenance, and analysis of synchronization behaviors, we recommend that all synchronization operations be syntactically bound to the corresponding lock object.

---

**Rule 4.23: Inherit from `PCP` in Any Class That Uses `PriorityCeilingEmulation MonitorControl` policy**

---

Developers should decide when the synchronization code is written whether it will use `PriorityCeilingEmulation` or `PriorityInheritance MonitorControl` policy. Code that intends to use `PriorityCeilingEmulation` should inherit from the `PCP` interface.

---

**Rule 4.24: Inherit from `Atomic` in Any Class That Synchronizes with Interrupt Handlers**

---

The `Atomic` interface extends the `PCP` interface. The byte-code verifier enforces that all `synchronized` methods belonging to classes that implement `Atomic` are `@StaticAnalyzable` and non-blocking in all execution modes.

---

**Rule 4.25: Do not inherit from PCP (or Atomic) if any ancestor class uses priority inheritance synchronization**

---

Any class file that includes a `synchronized` method but does not extend PCP is assumed to use priority inheritance in the implementation of the synchronization lock. To simplify semantics and implementation, we forbid mixing of priority inheritance synchronization and priority ceiling synchronization in any class or subclass definition.

---

**Rule 4.26: Annotate the `ceilingPriority()` method of Atomic and PCP Classes with `@Ceiling`**

---

The `@Ceiling` annotation expects its `value` attribute to be set to the ceiling priority at which this object expects to synchronize. Availability of an object's intended ceiling priority as a source code attribute makes it possible to prove compatibility between components using static analysis tools. In particular, the static analyzer can demonstrate that nested locks have strictly increasing ceiling priority values.

---

**Rule 4.27: Do Not Override `Object.finalize()`**

---

In traditional Java code, an object's `finalize()` method is invoked by the garbage collector before the object's memory is reclaimed. In the hard real-time domain, we do not have a garbage collector. Memory is reclaimed as particular control contexts are left. If finalization code is required, place it in the `finally` clause of a `try-finally` statement.

---

**Rule 4.28: Use Development Tools to Enforce Consistency With Hard Real-Time Guidelines**

---

To enforce that programmers make proper use of the hard real-time API subsets and that all code is consistent with the intent of the hard real-time programming annotations described in this section, use special byte-code verification tools that help assure reliable and efficient implementation of programmer intent.

## 5. Safety-Critical Development Guidelines

Safety-critical developers use a subset of the full hard real-time mission-critical capabilities.

---

**Rule 5.1: Except Where Indicated to the Contrary, Use Hard Real-Time Programming Guidelines**

---

In general, all of the hard real-time guidelines are appropriate for safety-critical development, except that certain practices acceptable for hard real-time mission-critical development should be avoided with safety-critical software.

---

**Rule 5.2: Use Only 28 Priority Levels for `NoHeapRealtimeThread`**

---

The official RTSJ specification states that a compliant implementation must provide at least 28 priorities, but may support many more. For safety-critical development, application software should limit its use of

**(Permission granted to reproduce and distribute as a complete document, without modification)**

priorities to the range from 1 through 28. Vendors can readily support this priority range as a standard safety-critical platform.

---

**Rule 5.3: Use Only Instances of `COMPLIANT-mode NoHeapRealtimeThread`**

---

In safety-critical systems, the sharing of data and control between native code and safety-critical Java code is strongly discouraged.

---

**Rule 5.4: Prohibit Use of `@OmitSubscriptChecking` Annotation**

---

In safety-critical code, turning off subscript checking is strongly discouraged, even though static analysis of the program presumably has proven that the program will not attempt to access invalid array elements. In safety-critical systems, the key benefit of subscript checking is to prevent an error in one component from propagating to other components.

---

**Rule 5.5: Prohibit Use of `@OmitScopeChecking` Annotation**

---

In safety-critical code, turning off assignment scope checking in methods that permit `@AllowCheckedScopedLinks` is strongly discouraged, even though careful inspection of the program may have determined that the program will not attempt to make assignments that would violate nested scoping rules.

---

**Rule 5.6: Prohibit Invocation of Methods Declared with `@AllowCheckedScopedLinks` Annotation**

---

This annotation is designed to allow programmers to use practices that cannot be certified safe by automatic static theorem provers. Thus, there is a risk that any software making use of this annotation will abort with a run-time exception. Allow this practice only in safety-critical systems for which developers are able to provide absolute proof that run-time exceptions will not be thrown.

---

**Rule 5.7: Require All Code To Be `@StaticAnalyzable`**

---

In hard real-time mission-critical code, the use of the `@StaticAnalyzable` annotation is entirely optional. In safety-critical code, we require all components to have this annotation, and for all relevant modes of analysis to have a `true` value for the `enforce_time_analysis`, `enforce_memory_analysis`, and `enforce_non_blocking` attributes.

---

**Rule 5.8: Require All Classes with Synchronized Methods to Inherit `PCP` or `Atomic`**

---

The safety-critical profile does not allow the use of priority inheritance locking.

(Permission granted to reproduce and distribute as a complete document, without modification)

---

**Rule 5.9: Prohibit Dynamic Class Loading**

---

While dynamic class loading may be supported in the hard real-time mission-critical domain, it should be strictly avoided in safety-critical software.

---

**Rule 5.10: Prohibit Use of Blocking Libraries**

---

Because of difficulties analyzing blocking interaction times when software components contend for shared resources, all services that might block are forbidden in safety-critical code. Specifically, the following APIs should not be invoked from safety-critical application software:

- `java.lang.Object.wait()`
- `java.lang.Object.wait(long)`
- `java.lang.Object.wait(long, int)`
- `java.lang.Thread.join()`
- `java.lang.Thread.join(long)`
- `java.lang.Thread.join(long, int)`
- `java.lang.Thread.sleep(long)`
- `java.lang.Thread.sleep(long, int)`
- `javax.realtime.util.mc.ThreadStack.join()`
- `javax.realtime.util.mc.CountingSemaphore.P()`
- `javax.realtime.util.mc.SignalingSemaphore.P()`
- `javax.realtime.util.mc.Mutex.enter()`

---

**Rule 5.11: Prohibit Use of PriorityInheritance MonitorControl policy**

---

Priority inheritance is more difficult to certify, more complicated to implement, and less efficient than priority ceiling emulation. Also, the implementation of priority inheritance introduces synchronization overhead delays that are proportional to the complexity of the application, rather than a function only of the system configuration. Analyzing these delays is particularly difficult. For all of these reasons, we prohibit its use in safety-critical software systems.

---

**Rule 5.12: Do Not Share Safety-Critical Objects With a Traditional Java Virtual Machine**

---

Combining safety-critical code with traditional Java code using the `@TraditionalJavaMethod` and `@TraditionalJavaShared` conventions compromises the integrity of the safety-certification artifacts. This practice is therefore strictly forbidden.

---

**Rule 5.13: Use Development Tools to Enforce Consistency With Safety-Critical Guidelines**

---

To enforce that programmers make proper use of the safety-critical subset and that all code is consistent with the intent of the hard real-time programming annotations described in this section, use special byte-code verification tools that help assure reliable and efficient implementation of programmer intent.

## 6. Enforcement of Style Guidelines

To empower development and maintenance productivity gains comparable to what is experienced by traditional Java developers, The byte-code verifier for hard real-time and safety-critical Java development enforces more constraints than are enforced by the traditional Java byte-code verifier. This section of the document details the additional byte-code verification constraints that are required for compliant implementations.

Byte-code verification is performed in two phases. Most byte-code verification is performed on individual classes, in isolation of the contents of other classes except for knowledge of their APIs. Certain byte-code verification must be deferred, however, until the complete program is assembled. We refer to these two phases of byte-code verification as *component verification* and *integration verification*.

### Component Verification

1. The determination of “legal program component” is based on analysis of individual classes and the methods they contain, in isolation of all other classes that might comprise a complete program. Byte-code verification is performed one class at a time.
  - a. The byte-code verification to determine whether a particular class is valid may require knowledge of the interface specifications for other classes with which this class interacts (by way of method invocations). Knowledge of another class’s interface specification includes both awareness of the class API and full knowledge of all meta-data annotation information associated with each API.
  - b. Additional static analysis beyond the byte-code verification required to determine whether particular program components are legal may be performed for the purposes of performance optimization. This analysis shall not affect the definition of legal program component. For example, there may be situations in which a particular composition of components is rejected as illegal because the “rules” of this standard require that the compiler generate run-time assignment checks but the program component is not annotated with the `@AllowCheckedScopedLinks` annotation. A more thorough static interprocedural analysis may reveal that no assignment checking is necessary. Assuming the code had been annotated with `@AllowCheckedScopedLinks`, it would be acceptable for a compliant implementation to elide the assignment checking instructions. However, it is not acceptable for a compliant implementation to treat the unannotated program component as “legal”.
2. Certain reference variables are known to the byte-code verifier as `@Scoped` variables. The byte-code verifier assures that the value of a `@Scoped` variable is never assigned to any variable that is not also known as a `@Scoped` variable. A reference variable is considered to be `@Scoped` if one or more of the following is true:
  - a. The variable is an incoming argument, and it is declared with the `@Scoped` or `@ScopedArray` annotation.

- b. The variable is an incoming argument, and the method is declared with the `@ScopedPure` annotation. If the argument represents an array of references, the argument is treated as a `@ScopedArray` variable.
  - c. The variable is named `this`, and the method or constructor is declared with the `@ScopedThis` or `@ScopedThisLocal` annotation.
  - d. The variable is named `this`, and the method or constructor is declared with the `@ScopedPure` annotation.
  - e. The variable is an instance variable, and is declared with the `@Scoped` or `@ScopedArray` annotation.
  - f. The variable is a static variable, and is declared with the `@Scoped` or `@ScopedArray` annotation.
  - g. The variable is unnamed (compiler generated), and holds the return result from a method invocation for a method that is declared with the `@Scoped` or `@ScopedArray` annotation.
  - h. The variable is an entry in an array of references, and the array itself is referenced from a variable that has the `@ScopedArray` annotation.
  - i. The variable is a local variable (either programmer declared, or automatically generated by the code generator), and the contents of this variable is never copied to any other variable that is not `@Scoped`. This means this variable's value is not passed to a constructor or method invocation as an argument that is not declared `@Scoped`, is not assigned to an instance or static field unless that field is declared as `@Scoped` or `@ScopedArray`, and it means that the variable's value is not returned from this method unless the method itself is declared with the `@Scoped` or `@ScopedArray` annotation.
3. A local variable (either programmer declared, or automatically generated by the code generator) is considered to have the `@ScopedArray` property if it represents an array of references and it satisfies all of the properties to be considered a `@Scoped` variable, and if none of the elements of this array are ever copied to an instance or static field, or to an outgoing argument, or returned from the method if the corresponding recipient of the copy operation is not labeled with the `@Scoped` or `@ScopedArray` attribute.
  4. A local `@Scoped` variable is considered `@ScopedLocal` if its value is never assigned to another variable that is not `@ScopedLocal`. A local `@ScopedArray` variable is considered `@ScopedArrayLocal` if its value is never assigned to another variable that is not `@ScopedArrayLocal`.
  5. Only reference variables may be characterized as `@Scoped` or `@ScopedArray` or `@ScopedLocal` or `@ScopedArrayLocal`.
  6. If a constructor is declared with the `@ScopedThis` attribute, it is understood that the constructed object may reside in a dynamic scope. A particularly difficult issue deals with incoming `@Scoped` arguments.
    - a. If a constructor is declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier assumes that incoming `@Scoped` arguments may potentially reside in scopes that are more inner-nested than the object that is being newly constructed. In this case, byte-code verification of the constructor's invocation does not assure that incoming `@Scoped` arguments originate in more outer-nested scopes.
    - b. In the case that a constructor with the `@ScopedThis` attribute does not include the `@AllowCheckedScopedLinks` annotation, the presumption of the byte-code verifier is that any incoming `@Scoped` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or

`@ScopedThisLocal` arguments) originate in a more outer scope than the object that is being constructed. This means:

- i. The byte-code verifier is responsible for assuring for any `@ScopedThis` constructor that does not specify the `@AllowCheckedScopedLinks` annotation at the point of the constructor's invocation that all `@Scoped` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or `@ScopedThisLocal` arguments) originate in a more outer-nested scope than the newly constructed object.
  - ii. If the newly constructed object is to reside in `ImmortalMemory`, the byte-code verifier is responsible for assuring that the `@Scoped` arguments passed to the constructor (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or `@ScopedThisLocal` arguments) also reside in `ImmortalMemory`.
  - iii. If static analysis of the method from which the constructor is invoked is not able to prove that the required preconditions are satisfied, run-time checks will be required at the point of the constructor's invocation. (If run-time checks are required at the point of the constructor's invocation, but this context is not declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier rejects the program.)
7. The byte-code verifier forbids the contents of a `@Scoped` variable from being assigned to a local variable, an instance or static field, an outgoing parameter, or returned from the method unless the destination of the assignment operation is also considered to have the `@Scoped` attribute.
  8. The byte-code verifier forbids the contents of a `@ScopedLocal` or `@ScopedThisLocal` variable from being assigned to another variable that is not also `@ScopedLocal`. Since only arguments and local variables may have this annotation, this means a `@ScopedLocal` variable can never be assigned to an instance or static field.
  9. The byte-code verifier forbids the contents of a `@ScopedArrayLocal` variable from being assigned to another variable that is not also `@ScopedArrayLocal`. Since only arguments and local variables may have this annotation, this means a `@ScopedArrayLocal` can never be assigned to an instance or static field.
  10. The byte-code verifier forbids the contents of a `@ScopedArray` array element from being assigned to a local variable, an instance or static field, an outgoing parameter, or returned from the method unless the destination of the assignment operation is also considered to have the `@Scoped` attribute.
  11. The byte-code verifier forbids the contents of a `@ScopedArray` variable from being assigned to a local variable, an instance or static field, an outgoing parameter, or returned from the method unless the destination of the assignment operation is also considered to have the `@ScopedArray` attribute.
  12. A return statement is treated as an assignment to an implicit outgoing return parameter. The return parameter is considered to have the `@Scoped` or `@ScopedArray` attribute only if the method is declared to have the `@Scoped` or `@ScopedArray` attribute respectively.
  13. Assignment of a `@Scoped` (or `@ScopedArray`) value to an instance or static field that has the `@Scoped` (or `@ScopedArray`) attribute requires in the most general case a run-time check to assure that the assigned value refers to an object that is more outer nested on the scope stack than the object that contains the field to be assigned. The run-time check consists of the following:
    - a. Check to see whether the value to be written holds a reference to a scope-allocated object. If not, no further checking is required.

- b. If so, check to see whether the object to be overwritten was also scope allocated. If not, throw a run-time exception because this violates the sharing protocol.
  - c. Otherwise, check to make sure the object to be overwritten does not reside in a context that is more outer nested than the object whose reference is to be assigned. If it does, throw a run-time exception because this violates the reference sharing protocol.
14. The byte-code verifier rejects any code that requires a run-time check unless the method that contains the code was declared to have the `@AllowCheckedScopedLinks` attribute. The following special cases do not require run-time checks and shall be allowed by the byte-code verifier even within methods that do not have the `@AllowCheckedScopedLinks` attribute:
  - a. If the object that contains the field to be assigned was just allocated within this thread's inner-most `ScopedMemory` context, we are assured that any values assigned to this field reside in the same or outer-nested contexts. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
  - b. If the value to be assigned was copied from another reference field (or array element) of the same object, we are assured that the assigned reference value refers to an object residing in the same or outer-nested context. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
  - c. If the value to be assigned was copied from another reference field (or array element) of an object that was reachable from the object that is being assigned to, we are assured that the assigned reference value refers to an object residing in the same or outer-nested context. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
  - d. If a reference value being returned from a method was passed in as an argument to the method, or was reachable from one of the objects passed in as arguments to the method, we are assured that the returned value is visible in the scope of the caller's method. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
  - e. Certain assignments made to `private` instance fields of classes declared with the `@ReentrantScope` or `@NestedReentrantScope` annotations need not be checked, as described in rule 43.
15. If a method is declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier allows assignments to instance and static fields even in circumstances that the byte-code verifier cannot guarantee that the object or class that contains the fields resides in a more outer-nested scope than the assigned value. It is presumed in these situations that the code generator will insert run-time checks to assure that each reference assignment is legal.
16. For any method declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier assures that the method is declared to throw `javax.realtime.util.mc.IllegalAssignmentException`, which is checked. Note that enforcement of this restriction also assures that all of the superclass methods that this method might override are also declared to throw `javax.realtime.util.mc.IllegalAssignmentException`.
17. A method that declares `@CallerAllocatedResult` must satisfy the following conditions:
  - a. For each `return` statement within the method, it is reached by exactly one defining expression that defines the value returned by this expression.
  - b. The allocating expression is of one of the four following forms:
    - i. `result_variable = new <object>();` where `<object>` represents the declared return type for this method, or one of the subtypes identified in the `subclasses` attribute associated with the

`@CallerAllocatedResult` annotation and the invoked constructor is declared with the `@ScopedThis` or `@ScopedPure` annotations; or

- ii. `result_variable = new <object>[N]`; where `<object>[]` represents the declared return type for this method or `<object>` represents one of the subtypes identified in the subclasses attribute associated with this method's `@CallerAllocatedResult` annotation. Immediately following this assignment statement, the byte-code verifier requires an annotation to specify the length of the allocated array. These annotations take one of the following two forms:

```
assert StaticLimit.ArrayLength(result_variable, Max);
```

where `Max` represents the maximum number of elements in the allocated array, or

```
assert StaticLimit.ArrayLength(result_variable, mode_1, Max_1);
assert StaticLimit.ArrayLength(result_variable, mode_2, Max_2);
...
assert StaticLimit.ArrayLength(result_variable, mode_N, Max_N);
```

where `Max_N` represents the maximum number of elements in the allocated array if the assertion is evaluated according to analysis `mode_N`; and this method was declared with the `@StaticAnalyzable` annotation, and the `modes` attribute of this annotation enumerates  $N$  analysis modes representing `mode_1`, `mode_2`, ... `mode_N`, or

- iii. `result_variable = methodCall()`; where `methodCall()` is declared to return the same type as this method, and it also has the `@CallerAllocatedResult` annotation and its `subclass` attribute represents a subset of the allowed subclass return types for this method; or
- iv. `result_variable = <non-scoped-variable>`; meaning the returned value, which resides in `ImmortalMemory`, is allocated independently of this method's execution. This mode is used to implement methods that sometimes need to return new objects allocated in the caller's context, but may alternatively return references to previously allocated `ImmortalMemory` objects, presumably representing certain "common" anticipated responses.

In the case that the return type is an array, the method must be declared with the `@StaticAnalyzable` annotation. Insofar as scoped memory allocation is concerned, it is not necessary for the `enforce_memory_analysis` attributes to be true. However, it is essential that every allocating expression provide enough `StaticLimit.ArrayLength()` assertions to fully constrain the array size for every possible execution mode corresponding to this method. This requirement is enforced by the byte code verifier.

- c. The allocating expression dominates the `return` statement. This means that every control path from this method's entry point to the particular `return` statement must pass through the identified allocating expression.

18. A method that declares `@CallerAllocatedArrayResult` must satisfy the following conditions:

- a. For each `return` statement within the method, it is reached by exactly one allocating expression that defines the value returned by this expression.
- b. The allocating expression is of one of the four following forms:
  - i. `result_variable = new <object>[N]`; where `<object>[]` represents the declared return type for this method or `<object>` represents one of the subtypes identified in the subclasses attribute associated with this method's `@CallerAllocatedResult` annotation. Immediately following this

assignment statement, the byte-code verifier requires an annotation to specify the length of the allocated array. These annotations take one of the following two forms:

```
assert StaticLimit.ArrayLength(result_variable, Max);
```

where **Max** represents the maximum number of elements in the allocated array, or

```
assert StaticLimit.ArrayLength(result_variable, mode_1, Max_1);
assert StaticLimit.ArrayLength(result_variable, mode_2, Max_2);
...
assert StaticLimit.ArrayLength(result_variable, mode_N, Max_N);
```

where **Max\_N** represents the maximum number of elements in the allocated array if the method is executed according to analysis **mode\_N**; and this method was declared with the **@StaticAnalyzable** annotation, and the **modes** attribute of this annotation enumerates *N* analysis modes representing **mode\_1**, **mode\_2**, ... **mode\_N**, or

- ii. **result\_variable = methodCall();** where **methodCall()** is declared to return the same type as this method, and it has the **@CallerAllocatedResult** attribute and its **subclass** attribute represents a subset of the allowed subclass return types for this method; or
- iii. **result\_variable = methodCall();** where **methodCall()** is declared to return the same type as this method, and it also has the **@CallerAllocatedArrayResult** attribute and its **subclass** attribute represents a subset of the allowed subclass return types for this method; or

In the case that the allocating expression is of the form described in paragraph 18.b.i, the enclosing method must be declared with the **@StaticAnalyzable** annotation. Insofar as scoped memory allocation is concerned, it is not necessary for the **enforce\_memory\_analysis** attributes to be true. However, it is essential that every allocating expression provide enough **StaticLimit.ArrayLength()** assertions to fully constrain the array size for every possible execution mode corresponding to this method. This requirement is enforced by the byte code verifier.

- c. The allocating expression dominates the **return** statement. This means that every control path from this method's entry point to this particular **return** statement must pass through the identified allocating expression.
- d. In the case that the allocating expression is of one of the first two forms (18.b.i or 18.b.ii), there must also exist a **for** loop of the following form which dominates the **return** statement relative to the allocating expression:

```
for (int i = 0; i < result_variable.length; i++) {
    // arbitrary code, not shown
    result_variable[i] = <allocating-expression>;
    // more arbitrary code, not shown
}
```

We require that the body of this loop not make any assignments (or use the increment/decrement operators) that would modify the value of the loop's counting variable *i*. Within this loop, there is a single assignment to the expression **result\_variable[i]**. This assignment is dominated by the first instruction of the loop's body and this expression dominates the last instruction in the loop's body.

In other words, every iteration of the loop executes the allocation expression and corresponding assignment exactly once. The allocating expression of one of the following two forms:

- i. `result_variable[i] = new <object>();` where `<object>` represents the declared element type for the `result_variable` array or one of the subclasses represented by the `subclass` attribute associated with this method's `@CallerAllocatedArrayResult` annotation and the invoked constructor is declared with the `@ScopedThis` or `@ScopedPure` annotation, or
- ii. `result_variable[i] = methodCall();` where `methodCall()` is declared to return the same type as the declared element type for the `result_variable` array and the invoked method is declared with the `@CallerAllocatedResult` annotation, and its `subclass` attribute represents a subset of the subclasses associated with this method's `@CallerAllocatedArrayResult` annotation.

The byte-code verifier assures that there are no other assignments to any of the result array's elements by requiring that:

- i. `result_variable` is not passed as an argument to any other methods before being returned from this method, and
- ii. `result_variable` is not copied to any other variables within this method, and
- iii. within this method, the only allowed use of the `result_variable[subscripting-expression]` subexpression as an *l-value* (a value to which it is possible to assign a value) is the single expression contained within the iteration loop described above.

19. A method that is declared with the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations is treated similarly to constructors that are declared with the `@ScopedThis` annotation. In particular:

- a. If the caller-allocated result method is declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier assumes that incoming `@Scoped` arguments may potentially reside in scopes that are more inner-nested than the object that is being newly constructed. In this case, the byte-code verifier does not assure that incoming `@Scoped` arguments originate in more outer-nested scopes.
- b. In the case that a caller-allocated result method is not declared using the `@AllowCheckedScopedLinks` annotation, the presumption of the byte-code verifier is that any incoming `@Scoped` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or `@ScopedThisLocal` arguments) originate in a more outer scope than the scope within which the caller-allocated result is to be constructed. This means:
  - i. The byte-code verifier is responsible for assuring for any caller-allocated result method that does not specify the `@AllowCheckedScopedLinks` annotation at the point of the method's invocation that all `@Scoped` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or `@ScopedThisLocal` arguments) originate in a more outer-nested scope than the scope that will hold the method's result.
  - ii. At the point of the outer-most invocation of a series of nested caller-allocated result methods, the byte-code verifier determines whether the result will reside in the local scope or within `ImmortalMemory`. If the byte-code verifier determines that the result will reside in `ImmortalMemory`, it takes responsibility for assuring that all `@Scoped` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` or `@ScopedThisLocal` arguments) passed to the caller-allocated result method reside in `ImmortalMemory`.

- iii. If static analysis of the method from which the caller-allocated result method is invoked is not able to prove that the required preconditions have been satisfied, then run-time checks will be required at the point of the method's invocation to assure that the method's arguments are internally consistent. Note that these run-time checks are only allowed if the invoking method is declared with the `@AllowCheckedScopedLinks` annotation.
  - c. If a caller-allocated result method returns the result of another caller-allocated result method and neither of the two methods is declared with the `@AllowCheckedScopedLinks` annotation, the byte-code verifier uses inductive reasoning to prove that the `@Scoped` arguments to the inner-nested invocation originate in outer-nested scopes. The inductive verification performed by the byte-code verifier assures the following:
    - i. The invoked method is declared with the same annotation as this method (either `@CallerAllocatedResult` or `@CallerAllocatedArrayResult`)
    - ii. The list of possible return subclass types associated with the invoked method is a subset of the allowed return types associated with this method.
    - iii. Any local variables to which the result of the invoked method might be assigned are treated as `@Scoped` variables.
    - iv. Any reference values passed as `@Scoped` arguments to the invoked caller-allocated result method originate outside this method. A value originates outside of a particular method if:
      - a. It was passed to the method as an incoming argument (either `@Scoped` or not), or
      - b. It was fetched from a reference field (either `@Scoped` or not) of an object referenced from another value originating outside this method, or
      - c. It was returned as a result from another method invocation, and that method's return result is not declared with the `@Scoped` annotation, or
      - d. It was allocated within this scope as an `ImmortalMemory` object.
      - e. It was copied from another variable that is not a `@Scoped` variable.
20. At the point of each `new` operation, the byte-code verifier determines whether the newly constructed object is to be allocated in some outer-nested allocation context, in the current method's allocation scope, or in `ImmortalMemory`. The rules are as follows:
- a. If the result of the `new` operation is assigned to a variable which is not a `@Scoped` variable or if the invoked constructor is not annotated with the `@ScopedThis` or `@ScopedPure` annotations, the new object will be allocated in `ImmortalMemory`.
  - b. Otherwise, if this method is declared with the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations, and there exists a data-flow path from the new operation's result to the method's return result, the new object is allocated within the caller's scope, using memory that was set aside by the caller to hold this method's result.
  - c. Because instantiation and manipulation of `StringBuilder` objects are often hidden within byte-code instructions that are automatically generated by the Java compiler, special handling is given to the treatment of `StringBuilder` objects. In the case that a locally instantiated `StringBuilder` object serves only:
    - i. to execute its own methods (e.g. `append()`, `length()`, `capacity()`, `ensureCapacity()`), and
    - ii. as an argument to invocation of `String(StringBuilder)` constructors, and

- iii. all `String` objects constructed from this `StringBuilder` object reside in the same scope, the `StringBuilder` object is allocated within the same scope that holds the corresponding `String` objects. If the `StringBuilder` object is used as an argument to construct multiple `String` objects residing in potentially different memory scopes, the `StringBuilder` is allocated in `ImmortalMemory`. The decision to allocate the `StringBuilder` object in `ImmortalMemory` will result in a byte-code verification error if the corresponding method is not annotated with the `@ImmortalAllocation` annotation.
- d. When objects are allocated within instance methods of `@ReentrantScope` or `@NestedReentrantScope` objects, special handling is provided:
  - i. If the byte-code verifier is able to assure that no reference to an allocated object survives beyond the method's activation, the object is allocated in the temporary scope of the method's activation frame. This determination is based on reachability analysis. If references to the allocated object only reach local variables or outgoing arguments that are declared `@ScopedLocal` or `@ScopedArrayLocal`, the object is allocated in the local activation frame.
  - ii. Otherwise, if the newly allocated object is assigned to a `@Scoped` argument, the new object is allocated within the shared reentrant scope of the enclosing object.
  - iii. Otherwise, the newly allocated object is allocated in `ImmortalMemory`.
- e. Otherwise, the memory for the new object is allocated within the current method's dedicated allocation context.

The byte-code verifier coordinates with the code generator and static analyzer to assure that the memory is allocated in the proper context, and the analysis of required sizes for each memory scope is updated appropriately.

- 21. Whenever the byte-code verifier determines that a particular memory allocation must be satisfied out of `ImmortalMemory`, it assures that the method that is responsible for requesting the memory allocation is declared with the `@ImmortalAllocation` annotation. If this annotation is missing, the byte-code verifier rejects the code. There are two situations in which a method is responsible for allocation of a new object:
  - a. In the case that a method includes a `new` operation, that method is responsible for allocation of the requested object.
  - b. In the case that a method includes an invocation of another method, and that other method is declared with the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations, this method is responsible for allocation of the object to be returned from the invoked method, except when this method itself was declared with the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations and the result returned from the invoked method is returned from this method.
- 22. If a method or constructor is declared with the `@StaticAnalyzable` annotation, this denotes that the body of the method or constructor must be analyzable by static analysis tools. The byte-code verifier consults the attributes of the `@StaticAnalyzable` annotation to determine which kinds of analysis are required. All of these attribute values are available to the byte-code verifier, because they are specified as constant values in the source code.
  - a. The byte-code verifier looks at the `modes` attribute to determine how many different modes of analysis are required for this method.

- b. The byte-code verifier requires that the `enforce_memory_analysis` array has either a single entry, or has one entry for each of the enumeration values represented by the `modes` attribute. The byte-code verifier consults the `enforce_memory_analysis` array to determine in which modes of execution the static analyzer must be able to determine the maximum amount of `ImmortalMemory` and local-scope memory that will be allocated during allocation of this method. If this array has only a single entry, the value of this entry corresponds to every analysis mode.
  - c. The byte-code verifier requires that the `enforce_time_analysis` array has either a single entry, or has one entry for each of the enumeration values represented by the `modes` attribute. The byte-code verifier consults the `enforce_time_analysis` array to determine in which modes of execution the static analyzer must be able to determine the maximum amount of `ImmortalMemory` and local-scope memory that will be allocated during allocation of this method. If this array has only a single entry, the value of this entry corresponds to every analysis mode.
  - d. The byte-code verifier requires that the `enforce_non_blocking` array has either a single entry, or has one entry for each of the enumeration values represented by the `modes` attribute. The byte-code verifier consults the `enforce_non_blocking` array to determine in which modes of execution the static analyzer is required to demonstrate that this method does not invoke any blocking operations. If this array has only a single entry, the value of this entry corresponds to every analysis mode.
23. For each of the analysis modes that are identified in the `@StaticAnalyzable` annotation, if the `@StaticAnalyzable` annotation identifies that the corresponding value of the `enforce_memory_analysis`, `enforce_time_analysis`, or `enforce_non_blocking` attributes are `true`, the byte code verifier assures that all annotations required to enforce the analysis are present. If not present, the byte code verifier rejects the program as illegal.
24. For any method that includes memory allocations that are to be satisfied from its dedicated memory scope, the byte-code verifier assures that:
- a. The method is declared with the `@StaticAnalyzable` annotation and the `enforce_memory_analysis` attribute is `true` for all analysis modes, or
  - b. The method is declared with the `@ScopedMemorySize` annotation, or
  - c. The method includes an invocation of `SizeEstimator.ensureScopeCapacity()`.
25. For every method that has the `@StaticAnalyzable` annotation, the byte-code verifier examines the body of the method to assure that the appropriate information can be statically determined. In particular:
- a. For each analysis mode corresponding to a `true` value of `enforce_time_analysis`, the byte-code verifier assures that:
    - i. Every loop within the program that is reachable according to this analysis mode is accompanied by a `StaticLimit.IterationBound()` or `StaticLimit.NestedIterationBound()` assertion to limit the number of times the loop iterates.
    - ii. Every new-array operation on any legal path through the program that corresponds to this particular execution mode must be followed immediately by a `StaticLimit.ArrayLength()` assertion (or by a sequence of assertions, each corresponding to a different execution mode for this method) specifying an upper bound on the size of the allocated array. This annotation bounds the time required to initialize the array data.

- iii. Every invoked method on every legal path corresponding to this analysis mode is itself declared with the `@StaticAnalyzable` annotation.
  - iv. Each invocation of a method is immediately preceded by a `StaticLimit.InvocationMode()` assertion (or a sequence of assertions, each corresponding to a different execution mode for this method) specifying the mode under which the invoked method is to be analyzed. The byte-code verifier assures that the argument to `StaticLimit.InvocationMode()` is a legal analysis mode of the invoked method, and further assures that the corresponding `enforce_time_analysis` entry for that invoked method is `true`. In the case that the invoked method has only one analysis mode, the `InvocationMode()` assertion is not required.
  - v. The method does not recursively invoke itself on any execution paths that are legal for this analysis mode.
- b. For each analysis mode corresponding to a `true` value of `enforce_memory_analysis`, the byte-code verifier assures that:
- i. Every loop within the program that contains at least one memory allocation request, and is reachable according to this analysis mode is accompanied by a `StaticLimit.IterationBound()` or `StaticLimit.NestedIterationBound()` assertion to limit the number of times the loop iterates.
  - ii. Every new-array operation on any legal path through the program that corresponds to this particular execution mode must be followed immediately by a `StaticLimit.ArrayLength()` assertion (or by a sequence of assertions, each corresponding to a different execution mode for this method) specifying an upper bound on the size of the allocated array.
  - iii. Every invoked method on every legal path corresponding to this analysis mode is itself declared with the `@StaticAnalyzable` annotation.
  - iv. Each invocation of a method is immediately preceded by a `StaticLimit.InvocationMode()` assertion (or a sequence of assertions, each corresponding to a different execution mode for this method) specifying the mode under which the invoked method is to be analyzed. The byte-code verifier assures that the argument to `StaticLimit.InvocationMode()` is a legal analysis mode of the invoked method, and further assures that the corresponding `enforce_memory_analysis` entry for that invoked method is `true`. In the case that the invoked method has only one analysis mode, the `InvocationMode()` assertion is not required.
  - v. The method does not recursively invoke itself on any execution paths that are legal for this analysis mode.
- c. For each analysis mode corresponding to a `true` value of `enforce_non_blocking`, the byte-code verifier assures that:
- i. Every invoked method on every legal path through the program that corresponds to this particular execution mode must itself be declared with `@StaticAnalyzable` annotation.
  - ii. Each invocation of a method is immediately preceded by a `StaticLimit.InvocationMode()` assertion (or a sequence of assertions, each corresponding to a different execution mode for this method) specifying the mode under which the invoked method is to be analyzed. The byte-code verifier assures that the argument to `StaticLimit.InvocationMode()` is a legal analysis mode of the invoked method, and further assures that the corresponding `enforce_non_blocking` entry for that invoked method is `true`. In the case that the invoked method has only one analysis mode, the `InvocationMode()` assertion is not required.

- iii. For every invoked `synchronized` method on every legal path through the program corresponding to this particular execution mode, the invoked `synchronized` method corresponds to a class that is declared to implement the `javax.realtime.util.sc.Atomic` interface.
  - iv. This method does not invoke `CountingSemaphore.P()` or `SignalingSemaphore.P()`, or `Mutex.lock()`.
  - v. This method does not invoke `Object.wait()`.
26. The byte-code verifier assures that the only fields of the `@StaticAnalyzable` meta-data annotation that are assigned by the developer are `modes`, `enforce_time_analysis`, `enforce_memory_analysis`, `enforce_non_blocking`, `user_estimated_time`, `user_estimated_heap`, and `user_estimated_stack`. All of the other fields, which are named `execution_time`, `stack_bytes`, `heap_bytes` contain their default values.
  27. The enumeration class supplied as the argument to the `@StaticAnalyzable.modes` attribute must have at least one element.
  28. The arguments to all `StaticLimit` assertions are compile-time constants.
  29. Every `StaticLimit.IterationBound()` assertion is contained within a loop.
  30. Every `StaticLimit.NestedIterationBound()` assertion is contained within an appropriately nested loop.
  31. Any mode selection parameter (`caller_mode`) passed to `StaticLimit` assertion enforcement is of the type corresponding to the enclosing method's `StaticAnalyzable` annotation.
  32. Each `StaticLimit.InvocationMode()` assertion specifies an analysis mode for the invoked method which is one of the enumeration constants associated with the `@StaticAnalyzable` annotation for the invoked method.
  33. Every occurrence of a `StaticLimit.InvocationMode()` assertion immediately precedes an invocation of a method.
  34. In any method that includes an invocation of `SizeEstimator.ensureScopeCapacity()`, the following rules are enforced:
    - a. If this is not a `static` method, the class is not declared with the `@ReentrantScope` or `@NestedReentrantScope` annotations.
    - b. The method is not declared with the `@StaticAnalyzable` or `@ScopedMemorySize` annotations.
    - c. The method contains only one invocation of `ensureScopeCapacity()`.
    - d. The invocation of `ensureScopeCapacity()` dominates all allocations within the method's local scope, except for a single optional instantiation of a `@Scoped SizeEstimator` object.
  35. In any constructor that includes an invocation of `SizeEstimator.ensureScopeCapacity()`, the following rules are enforced:
    - a. The class is declared with the `@ReentrantScope` or `@NestedReentrantScope` annotations.
    - b. The class is not declared with the `@StaticAnalyzable` or `@ScopedMemorySize` annotations.
    - c. The constructor is not declared with the `@StaticAnalyzable` or `@ScopedMemorySize` annotations.
    - d. The constructor contains only one invocation of `ensureScopeCapacity()`.

- e. The invocation of `ensureScopeCapacity()` dominates all allocations within the constructor's local scope, except for a single optional instantiation of a `@Scoped SizeEstimator` object.
36. For any class that is declared with the `@ReentrantScope` or `@NestedReentrantScope` annotations, the byte-code verifier assures the following:
- a. If the class itself is declared to have the `@StaticAnalyzable` attribute, each constructor must also be declared with the `@StaticAnalyzable` attribute and the analysis modes for each constructor and for the class itself must be identical.
  - b. If the class is declared with the `@StaticAnalyzable` attribute, then for all analysis modes for which `enforce_memory_analysis` is `true`, the byte-code verifier assures that:
    - i. Every instance method that allocates memory is declared with the `@StaticAnalyzable` annotation and `enforce_memory_analysis` is `true` for all of the method's analysis modes.
    - ii. Every instance method that allocates memory is annotated with one `StaticLimit.Invocation-Bound()` assertion for each of the class's allowed analysis modes that dominates all exits from the method.
37. For every class declared with the `@NestedReentrantScope` annotation, the byte-code verifier assures that:
- a. This class is only instantiated from within a constructor of an object that is declared either with the `@ReentrantScope` or `@NestedReentrantScope` annotations.
  - b. No instance of this class is ever coerced to a superclass type.
  - c. For any instance method that is declared to return a `@Scoped` or `@ScopedArray` reference value, the byte-code verifier assures that every one of the reaching definitions matches one of the following patterns:
    - i. The return value is copied from a private variable that is within the object's *safe-set*, or
    - ii. The return value can legitimately be assigned to a variable within this object's *safe-set*, as characterized by the patterns described in rule 43.b.ii.
38. For each instantiation of a class declared with the `@ReentrantScope` or `@NestedReentrantScope` annotations, the byte-code verifier takes responsibility for determining whether the object will be allocated in `ImmortalMemory` or within scoped memory. The object will be allocated in scoped memory if and only if all of the following conditions are satisfied. Otherwise, it will be allocated in `ImmortalMemory`.
- a. The invoked constructor must be declared with the `@ScopedThis` or `@ScopedPure` annotations.
  - b. The variable that is assigned the result of construction must be a `@Scoped` variable.
39. The byte-code verifier assumes that any incoming `@Scoped` or `@ScopedArray` values passed as arguments to the instance methods of a `@ReentrantScope` object may potentially reside in scopes that are more inner-nested than the reentrant scope corresponding to this object's memory. Thus, the byte-code verifier assures the following:
- a. The incoming `@Scoped` argument's value is not assigned to any instance fields of existing or newly allocated objects, unless
  - b. The method is declared with the `@AllowCheckedScopedLinks` attribute, in which case the assignments are allowed, but must be accompanied by run-time checks to enforce scoping restrictions.

Note that an incoming `@Scoped` argument may be safely copied, with the new copy residing within the reentrant scope.

40. The byte-code verifier assures that any incoming `@Scoped` values and `@ScopedArray` values and individual array elements of `@ScopedArray` values which are passed as arguments to the instance methods of a `@NestedReentrantScope` object reside at the same scope level as the reentrant scope associated with the invoked instance method. If this fact cannot be verified from analysis of the method's body, the byte-code verifier assures that a run-time check is performed at the method's invocation point. If run-time checks are not allowed (because the invoking method does not specify `@AllowCheckedScopedLinks`), the byte-code verifier rejects the code.
41. If a `@ReentrantScope` or `@NestedReentrantScope` class is annotated with the `@StaticAnalyzable` annotation, the byte-code verifier assures that the default values of the `enforce_time_analysis` and `enforce_non_blocking` attributes are not overwritten.
  - a. Upon further reflection, Kelvin is not confident that this particular restriction is correct. The expectation is that a class might be declared `@StaticAnalyzable` because the developer wants to automatically determine the size of the reentrant-scope for use within the object's constructor. I think a more reasonable interpretation would be:
    - i. To consider that the connotation of annotation a class as `@StaticAnalyzable` means simply that the memory requirements of the class are statically analyzable.
    - ii. To allow each method to specify independently whether it enforces time analysis or non-blocking behavior.
    - iii. To require that each instance method of the class be declared as `@StaticAnalyzable` and that it enforce memory analysis for at least the same analysis modes as the class declaration itself.
42. For each class declared with a `@ReentrantScope` or `@NestedReentrantScope` annotation, the byte-code verifier assures that:
  - a. The class is declared as `@StaticAnalyzable`, and each instance method is declared as `@StaticAnalyzable` with the `enforce_memory_analysis` attribute `true` for all analysis modes, and each such method includes an invocation of `StaticLimit.InvocationBound()` which is not contained within any loops of the method and which dominates all exits from the method, or
  - b. The class is not declared as `@StaticAnalyzable` or the class is declared with `@StaticAnalyzable` annotation but at least one of the analysis modes has `enforce_memory_analysis` set to `false`, and the class is declared with a `@ScopedMemorySize` annotation, or
  - c. The class is not declared with either `@StaticAnalyzable` or `@ScopedMemorySize` annotations. In this case, the verifier assures that every constructor contains exactly one invocation of `javax.real-time.util.sc.SizeEstimator.ensureScopeCapacity()` which dominates every exit from the constructor.
43. We require that the byte-code verifier implement the following algorithm in analyzing `@ReentrantScope` and `@NestedReentrantScope` classes and all private inner classes:
  - a. Initially, consider all private `@Scoped` instance variables to be members of the *safe-set*.
  - b. Repeat until the inner loop completes without making any changes to the *safe-set* membership:
    - i. For each assignment to a variable in the *safe-set*, consider all of the reaching definitions.
    - ii. If any value assigned to this variable does not match at least one of the following descriptions, remove this variable from the *safe-set*:

- The value originates as a new-memory allocation within an instance method of the `@ReentrantScope` or `@NestedReentrantScope` class, or
- The value is a copy from another private variable within the enclosing object's *safe-set*, or
- The value is a copy of a `@Scoped` or `@ScopedArray` value returned from an instance method of a `@NestedReentrantScope` object, or
- The value is a copy of an array element and the array that contains this element is referenced from a private `@ScopedArray` instance variable that is contained within the *safe-set*, or
- The value is an incoming `@Scoped` argument to an instance method of a `@NestedReentrantScope` object.

Note that we prohibit direct assignment of an array element referenced from a local variable that was not copied from a *safe-set* instance variable. This means a `@ScopedArray` object returned from an instance method of a `@NestedReentrantScope` object must first be assigned to a *safe-set* private variable before its array element values can be safely fetched. (*Editor's note: it may not be possible to enforce this restriction, depending on what liberties the Java source compiler might take in optimization of source code. Revise this guideline if necessary.*)

- iii. Treat each array element of a `@ScopedArray` array which is referenced from a variable within the *safe-set* as a variable within the *safe-set*. If application of rule ii requires that the array element be removed from the *safe-set*, then the private variable that references this array is removed from the *safe-set*.

For any assignment to variables within the *safe-set*, the generated code shall perform no scoped-memory assignment checking. Furthermore, the byte-code verifier shall not require the `@AllowCheckedScopedLinks` annotation to accompany such assignments.

44. Within `@ReentrantScope` and `@NestedReentrantScope` classes, the byte-code verifier assures that the array elements contained within *safe-set* `@ScopedArray` objects are never overwritten with values that refer to memory external to this object's reentrant scope. Specifically, no value copied from a *safe-set* `@ScopedArray` variable is ever copied to another variable that is not also considered a *safe-set* `@ScopedArray` variable. Within the context of enforcing this particular rule, returning a reference to the *safe-set* `@ScopedArray` object is acceptable if and only if the value is returned from an instance method of a `@NestedReentrantScope` object and the instance method is declared with the `@ScopedArray` annotation. Note that it is specifically forbidden to copy the value of a *safe-set* `@ScopedArray` variable to a *safe-set* variable that does not carry the `@ScopedArray` attribute.
45. Within `@ReentrantScope` and `@NestedReentrantScope` classes, the byte-code verifier assures that all invocations of any `@NestedReentrantScope` instance methods satisfy the required argument passing conventions described in rule 40. Because the byte-code verifier is not able to enforce compliance with argument passing conventions within other contexts, the byte-code verifier assures that no reference to a `@NestedReentrantScope` object is ever copied to a variable that is not contained within the corresponding object's *safe-set*. This enforcement is based on analysis of reaching definitions.
46. If a class definition includes a variable declared with the `@Delegator` annotation, that variable must satisfy the following conditions:
  - a. It is an instance variable.
  - b. It is declared `final`.
  - c. It is the only variable declared with the `@Delegator` annotation within the class.

- d. It is initialized within every constructor by copying the value of that constructor's single argument that is declared with the `@Delegator` annotation.
  - e. Each constructor for the class has only one variable declared with the `@Delegator` annotation.
  - f. The type of each constructor argument that is declared with the `@Delegator` annotation is exactly the same as the type declared for the `@Delegator` instance variable.
47. If a class definition includes one or more variables declared with the `@Delegate` annotation, we enforce that:
- a. Each such variable is an instance variable.
  - b. Each such variable is declared `final`.
  - c. Each such variable is initialized within every constructor for the class by copying the value returned by a `new` operation for a class of the variable's declared type (not a subclass).
  - d. The invocation of the constructor for the `new` operation whose result is assigned to a `@Delegate` variable passes this as the value of the constructor's `@Delegator` argument.
48. If a method is declared `@DelegatedAnalyzable`, the verifier enforces the following:
- a. Either the class that contains the method has a final instance variable declared with the `@Delegator` annotation, or at least one of the method's arguments is declared with the `@Delegator` annotation, or both.
  - b. All of the annotations required for static analysis are present, enforcing the same restrictions as are imposed upon methods declared with the `@StaticAnalyzable` annotation, except that we do not necessarily require invocations of virtual methods that are associated with `@Delegator` variables to be analyzable. In particular, if a method is declared as `@DelegatedAnalyzable`, the byte-code verifier assures that the method's behavior can be analyzed in full accordance with the values of its `modes`, `enforce_time_analysis`, `enforce_memory_analysis`, and `enforce_non_blocking` attributes, conditioned upon resolution of the delegator's behavior with respect to these same attributes. The analysis of a `@DelegatedAnalyzable` method assumes that the `@Delegator` method's virtual methods will be analyzable in certain, but not necessarily all, execution contexts.
49. If analysis of a `@StaticAnalyzable` method depends on analysis of a `@DelegatedAnalyzable` method invoked from the original method, the byte-code verifier assures that the `@DelegatedAnalyzable` method is fully resolved from within the context of the invoking method. Specifically, the verifier assures that:
- a. The `@DelegatedAnalyzable` method is a `final` method, or
  - b. The `@DelegatedAnalyzable` method is a `static` method, or
  - c. The `@DelegatedAnalyzable` method is associated with a `@Delegate` instance variable.
50. To enforce scalable evolution and composition of software components, the byte-code verifier enforces certain consistency of annotations between related software components:
- a. If a particular method is declared with the `@CallerAllocatedResult` annotation, any overriding method must also declare the `@CallerAllocatedResult` annotation. Furthermore, the `subclasses` list associated with the overriding method must contain a (possibly improper) subset of the `subclasses` list associated with the original method.
  - b. If a particular method is declared with the `@CallerAllocatedArrayResult` annotation, any overriding method must also declare the `@CallerAllocatedArrayResult` annotation. Furthermore, the

`subclasses` list associated with the overriding method must contain a subset of (possibly equal to) the `subclasses` list associated with the original method.

- c. If a particular method is declared with the `@Scoped` annotation associated with its return value, all superclass methods that this method overrides must also declare the `@Scoped` annotation for their return values.
- d. If a particular method is declared with the `@ScopedThis` annotation, all overriding methods must also declare the `@ScopedThis` annotation.
- e. If a particular method provides the `@Scoped` annotation for one of its arguments, all overriding methods must also declare the `@Scoped` annotation for the same argument.
- f. If a particular method is declared with the `@ScopedPure` annotation, all overriding methods must also be declared with the `@ScopedPure` annotation.
- g. If a particular method is declared with the `@ImmortalAllocation` annotation, all superclass methods that this method overrides must also be declared with the `@ImmortalAllocation` annotation.
- h. If a particular method is declared with the `@AllowCheckedScopedLinks` annotation, all superclass methods that this method overrides must also be declared with the `@AllowCheckedScopedLinks` annotation.
- i. If a particular class is declared with the `@ReentrantScope` annotation, all subclasses must also be declared with the `@ReentrantScope` annotation. If the original `@ReentrantScope` class is declared with the `@StaticAnalyzable` annotation, all subclasses must also be declared with the `@StaticAnalyzable` annotation, with the same value for the `modes` attribute. Subclasses may impose greater enforcement of static analysis than the superclass by increasing the number of `true` entries in the `enforce_memory_analysis`, `enforce_time_analysis`, and/or `enforce_non_blocking` attributes. The subclass may not replace any `true` entries with `false` entries.
- j. If a particular class is declared with the `@NestedReentrantScope` annotation, all subclasses must also be declared with the `@NestedReentrantScope` annotation. If the original `@NestedReentrantScope` class is declared with the `@StaticAnalyzable` annotation, all subclasses must also be declared with the `@StaticAnalyzable` annotation, with the same value for the `modes` attribute. Subclasses may impose greater enforcement of static analysis than the superclass by increasing the number of `true` entries in the `enforce_memory_analysis`, `enforce_time_analysis`, and/or `enforce_non_blocking` attributes. The subclass may not replace any `true` entries with `false` entries.
- k. `@StaticAnalyzable`: If a given method is declared to be `@StaticAnalyzable`, then all overriding implementations of the same method must also be declared with the `@StaticAnalyzable` annotation and must have the same analysis modes. An overriding method may impose more stringent checking than was required by the superclass. For example, even though a particular method does not have the `enforce_memory_analysis` attribute set to `true` for a particular execution mode, an overriding subclass method may set this value to `true`.

If a new class is dynamically loaded, and the new class provides overriding implementations of previously loaded `@StaticAnalyzable` methods, and the overriding method's static resource requirements (either CPU-time or memory requirements) are greater than the static resource requirements of the previously analyzed superclass methods, the byte-code verifier rejects the dynamically loaded class. This is because we cannot reliably modify resource allocation decisions that had been based on the results of previous analysis.

Additionally, if a new class is dynamically loaded, and the new class provides overriding implementations of previously loaded `@StaticAnalyzable` methods, and the new methods introduce the possibility of indirect recursion among methods that are declared with the `@StaticAnalyzable` annotation, the byte-code verifier rejects the dynamically loaded class.

51. For any class that implements the `javax.realtime.util.Reconstructable` interface, the verifier assures that the class includes a public constructor that is declared with the `@StaticAnalyzable` and `@ScopedPure` annotations which does not have the `@AllowCheckedScopedLinks` annotation and does not throw any checked exceptions, and further verifies that none of the default attribute values associated with the `@StaticAnalyzable` annotation are overwritten in the declaration. The byte-code verifier further requires that this constructor take as a single argument an object of the same type as the class itself.
52. There is only one assignment statement to initialize the value of each static (class) variable, and this assignment is outside of any looping or conditional control structures. The assignment is either part of the variable's declaration or is contained within the body of a static initializer block.
53. All static initialization code is restricted according to the following rules:
  - a. The code is not allowed to perform any side effects (no synchronization, no I/O, no increment or decrement operations, and no assignments) except for the following:
    - i. A single assignment appearing outside of any looping control structures must be provided for each class variable.
    - ii. Assignments are allowed to the instance fields of objects that were allocated within the current scope and which are not reachable from any class variable (i.e. temporary objects).
    - iii. Assignments are allowed to the local variables of a static initialization context.
  - b. Only `final` methods may be invoked from within static initialization code.
54. For each static initializer block, the linker examines the byte code of the static initializer to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies) and which class variables are initialized by execution of this static initializer block. If this block does not initialize any class variables, the block is considered dead and a warning is issued by the compiler.
55. For each use of the `@StaticDependency` annotation, the identified class is available as part of the current class path and the named field is a static class variable associated with the identified class.
56. For each use of the `@StaticDependency` or `@InitializeAtStartup` annotations, the associated field declaration describes a static class variable.
57. Every invocation of `ThreadStack.spawn()` is contained within a `try` statement, and the corresponding `finally` statement includes a matching invocation of `ThreadStack.join()` for the same `ThreadStack` object.
58. The byte-code verifier assure that every invocation of `javax.realtime.util.mc.ThreadStack.spawn()` occur within the body of a `try` statement, and furthermore requires that the accompanying `finally` statement includes an invocation of `javax.realtime.util.mc.ThreadStack.join()`. To facilitate this analysis, the byte-code verifier requires that the `spawn()` invocation dominates all exits from the `try` statement, and requires that the corresponding `join()` invocation dominates all exits from the `finally` statement. Also, the byte-code verifier requires that there be no assignments to the variable that represents the `ThreadStack` object on any control path, including iterative control paths, between the `spawn()` and `join()` invocations.

59. The byte-code verifier assures that every method that is declared with the `@TraditionalJavaShared` annotation includes an invocation of `awaitClearRegistry()` as the last statement in the `finally` clause corresponding to the `try` clause that contains all executable code except the clean up code associated with the method. The clean up code, all of which must appear in the `finally` clause, consists only of:
  - a. Invocations of `ThreadStack.join()`,
  - b. Invocations of `Registry.unpublish()`, and
  - c. A single invocation of `Registry.instance().awaitClearRegistry()`. If this method's `@TraditionalJavaShared` annotation specifies the name of a particular Java virtual machine, the byte-code verifier assures that the same name is supplied as an argument to this `instance()` invocation.
60. A method declared with the `@TraditionalJavaMethod` annotation only accesses instance fields of its own object (`this`), static variables associated with its own class, and the elements of arrays that are referenced from its local, instance, or class static variables. It is not allowed to access instance fields of other objects or static fields of other classes.
61. Every method invoked from within a method declared with the `@TraditionalJavaMethod` annotation must be declared with the `@TraditionalJavaMethod` attribute.
62. Every method declared with the `@TraditionalJavaMethod` annotation must be declared with the `@ScopedPure` annotation and must not have the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations.
63. For any method that is annotated with the `@Ceiling` annotation, the byte-code verifier requires that the method's signature be:

```
public int ceilingPriority();
```

and further requires that the enclosing class implement the `javax.realtime.util.sc.PCP` interface. The byte-code verifier requires that the `value` attribute of this annotation be between 1 and 128 inclusive.
64. The byte-code verifier assures that every overriding implementation of `javax.realtime.util.sc.InterruptHandler.handleAsyncEvent()` is declared with the `@StaticAnalyzable` annotation, and furthermore requires that the `heap_bytes` attribute of the `@StaticAnalyzable` object equals zero.
65. The byte-code verifier assures that the program has no `synchronized` statements. It is only allowed to have `synchronized` methods.
66. All invocations of `Object.wait()`, `Object.notify()`, and `Object.notifyAll()` must reside within methods that are declared to be `synchronized`, and must apply to `this`.
67. The byte-code verifier assures that for any class that implements the `PCP` or `Atomic` interfaces, none of the superclasses which do not implement `PCP` interface has any `synchronized` methods.
68. The byte-code verifier assures that the program does not instantiate any occurrences of `java.lang.Thread` or `javax.realtime.RealtimeThread`.
69. The byte-code verifier assures that all `final` variables associated with a particular newly constructed object have been initialized within a constructor before the constructor assigns the value of `this` to any object fields or passes the value of `this` as an argument to another method.
70. The byte-code verifier assures that the argument to a `throw` statement is not a `@Scoped` variable. (Only exceptions residing in `ImmortalMemory` are allowed to be thrown.) (If we subsequently decide to relax this rule, then we need to introduce the `ThrowBoundaryError` exception and provide appropriate scope checking in the implementation of the `throw` operation.)

## Integration Verification

1. For every method that has the `@StaticAnalyzable` annotation, the byte-code verifier examines the body of the method to assure that the appropriate information can be statically determined. In particular:
  - a. For each analysis mode corresponding to a `true` value of `enforce_time_analysis`, the byte-code verifier performs a call-chain analysis of all methods that might be invoked (either directly or indirectly) from this method on control paths that are legal for this analysis mode in order to assure that there is no recursion.
  - b. For each analysis mode corresponding to a `true` value of `enforce_memory_analysis`, the byte-code verifier performs a call-chain analysis of all methods that might be invoked (either directly or indirectly) from this method on control paths that are legal for this analysis mode in order to assure that there is no recursion.
2. The integration verifier takes responsibility for assuring that all static variables are initialized, and that there are no circularities or ambiguities in the definition of static variables. It performs the following analysis:
  - a. For each declared class, the compiler separates the class variable initialization declarations into one expression for each variable to be initialized. The linker analyzes the byte code of each assignment expression to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies).
  - b. For each compound static class initialization statement, the compiler analyzes the body of the compound statement to determine which class variables it initializes and which class variables it depends on. The compound class initialization statement is treated as an indivisible unit. This block of code will be scheduled for execution after all dependencies have been resolved. Once executed, all class variables that are defined by this initialization statement are considered to be resolved.
  - c. The byte-code verifier makes no attempt to analyze the dependencies or side effects of native methods that are invoked in static initialization contexts.
  - d. For any static variable declaration that is annotated with a `@StaticDependency` annotation, the byte-code verifier adds the named dependency to its own analysis of static variables upon which this variable's initialization expression depends.
  - e. If a `@StaticDependency` annotation indicates that a particular variable depends on another variable, the initialization expression for that other variable includes native code, and the byte-code verifier is not able to find any Java initialization of this variable, the byte-code verifier assumes that the same native code initializes this variable.
  - f. If there is no initialization code for any static variables, the integration verifier rejects the program, listing the variables that could not be initialized.
  - g. Based on the dependency analysis for the complete set of classes associated with a particular program, the integration verifier performs a topological sort on all initialization components. If the topological sort cannot determine a complete ordering for assignments, the integration verifier rejects the program, noting which variables could not be safely initialized because of circular initialization dependencies.

## Situational Verification

Different teams and different projects will desire to enforce particular style guidelines on specific aspects of their development efforts. We recommend that byte-code verification tools have the ability to optionally enforce all of the following restrictions:

1. Disallow any use of the `@AllowCheckedScopedLinks` annotation.
2. Disallow any use of the `@ImmortalAllocation` annotation.
3. Require all methods to have the `@StaticAnalyzable` annotation.
  - a. Require all methods with the `@StaticAnalyzable` annotation to set the `enforce_memory_analysis` attribute to true for all analysis modes.
  - b. Require all methods with the `@StaticAnalyzable` annotation to set the `enforce_time_analysis` attribute to true for all analysis modes.
  - c. Require all methods with the `@StaticAnalyzable` annotation to set the `enforce_non_blocking` attribute to true for all analysis modes.
4. Disallow user-provided resource estimates by prohibiting application developers from overwriting the default values of the `user_estimated_stack`, `user_estimated_heap`, and `user_estimated_time` attributes of `@StaticAnalyzable` and `@DelegatedAnalyzable` annotations.
5. Disallow any use of the `@OmitSubscriptChecking` annotation.
6. Disallow any use of the `@OmitScopeChecking` annotation.
7. Disallow instantiation of `NoHeapRealtimeThread` with the `mode` parameter equal to `NATIVE`.
8. Disallow any invocations of `javax.realtime.util.sc.NoHeapRealtimeThread.condescend()`.
9. Disallow any invocations of native methods.
10. Disallow all synchronized methods unless the class implements `Atomic`.
11. Disallow all synchronized methods unless the class implements `PCP`.
12. Prohibit all use of `@TraditionalJavaShared` annotations.
13. Prohibit all use of `@TraditionalJavaMethod` annotations.
14. Restrict the range of legal `@Ceiling` value attributes to 1-28 inclusive (for compatibility with the safety-critical profile)
15. Prohibit all invocations of `SizeEstimator.ensureScopeCapacity()`
16. Enforce that the only `javax.realtime` libraries invoked are from the approved subset described in this document.
17. Enforce that the only `java.lang` libraries invoked are from the approved subset described in this document.

## Appendix A: Hard Real-Time Subset of the J2SE API

### 1. Subset of the JDK 1.5 `java.lang` Package

Note that both the number of classes and the number of methods supported by each class have been reduced significantly from the official Java standard. To reduce space and formatting effort, this subset description is provided only in terms of raw Java type declarations. The semantic descriptions of the supported classes and methods are available in standard Sun documentation.

---

---

#### **ArithmeticException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class ArithmeticException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis ArithmeticException();
    public @StaticAnalyzable @ScopedPure ArithmeticException(String msg);
}
```

---

---

#### **ArrayIndexOutOfBoundsException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class ArrayIndexOutOfBoundsException extends java.lang.IndexOutOfBoundsException {
    public @StaticAnalyzable @ScopedThis ArrayIndexOutOfBoundsException();
    public @StaticAnalyzable @ScopedPure ArrayIndexOutOfBoundsException(String msg);
}
```

---

---

#### **ArrayStoreException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class ArrayStoreException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis ArrayStoreException();
    public @StaticAnalyzable @ScopedPure ArrayStoreException(String msg);
}
```

---

---

## AssertionError.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.StaticAnalyzable;

public class AssertionError extends Error
{
    public @StaticAnalyzable @ScopedThis AssertionError();
    public @StaticAnalyzable @ScopedPure AssertionError(String msg);
}
```

---

---

## Boolean.java

---

```
package java.lang;

public class Boolean implements Comparable<Boolean>
{
    public static Boolean FALSE = false;
    public static Boolean TRUE = true;
    public static Class<Boolean> TYPE = boolean.class;

    @ScopedThis public Boolean(boolean v);
    @ScopedPure public Boolean(String str);
    @ScopedThis public boolean booleanValue();
    @ScopedPure public int compareTo(Boolean b);
    @ScopedPure public boolean equals(Object obj);
    public static boolean getBoolean(@Scoped String str);
    @ScopedThis public int hashCode();
    public static boolean parseBoolean(@Scoped String str);
    @ScopedThis @CallerAllocatedResult public String toString();
    @CallerAllocatedResult public static String toString(boolean value);
    public static Boolean valueOf(@Scoped String str);
    public static Boolean valueOf(boolean b);
}
```

---

---

## Byte.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.Scoped;

import javax.realtime.util.sc.CallerAllocatedResult;

public class Byte extends Number implements Comparable<Byte>
{
```

```

public static final Class TYPE;

public static final byte MAX_VALUE;
public static final byte MIN_VALUE;

@ScopedThis public Byte(byte val);
@ScopedThis public Byte(String str) throws NumberFormatException;
@ScopedThis public byte byteValue();
@ScopedPure public int compareTo(Byte other);
@CallerAllocatedResult public static Byte decode(@Scoped String str)
    throws NumberFormatException;
@ScopedThis public double doubleValue();
@ScopedPure public boolean equals(Object obj);
@ScopedThis public float floatValue();
@ScopedThis public int hashCode();
@ScopedThis public int intValue();
@ScopedThis public long longValue();
public static byte parseByte(@Scoped String str) throws NumberFormatException;
public static byte parseByte(@Scoped String str, int base) throws NumberFormatException;
@ScopedThis public short shortValue();
@CallerAllocatedResult @ScopedThis public String toString();
@CallerAllocatedResult public static String toString(byte v);
@CallerAllocatedResult public static Byte valueOf(@Scoped String str)
    throws NumberFormatException;
@CallerAllocatedResult public static Byte valueOf(@Scoped String str, int base)
    throws NumberFormatException;
}

```

---

## Character.java

---

```

package java.lang;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

import javax.realtime.util.sc.CallerAllocatedResult;

public final class Character implements Comparable<Character>
{
    public static final int MIN_RADIX;
    public static final int MAX_RADIX;
    public static final char MIN_VALUE;
    public static final char MAX_VALUE;
    public static final Class<Character> TYPE;

    public static final byte UNASSIGNED;
    public static final byte UPPERCASE_LETTER;
    public static final byte LOWERCASE_LETTER;
    public static final byte TITLECASE_LETTER;
    public static final byte MODIFIER_LETTER;
    public static final byte OTHER_LETTER;
    public static final byte NON_SPACING_MARK;
}

```

```
public static final byte ENCLOSING_MARK;
public static final byte COMBINING_SPACING_MARK;
public static final byte DECIMAL_DIGIT_NUMBER;
public static final byte LETTER_NUMBER;
public static final byte OTHER_NUMBER;
public static final byte SPACE_SEPARATOR;
public static final byte LINE_SEPARATOR;
public static final byte PARAGRAPH_SEPARATOR;
public static final byte CONTROL;
public static final byte FORMAT;
public static final byte PRIVATE_USE;
public static final byte SURROGATE;
public static final byte DASH_PUNCTUATION;
public static final byte START_PUNCTUATION;
public static final byte END_PUNCTUATION;
public static final byte CONNECTOR_PUNCTUATION;
public static final byte OTHER_PUNCTUATION;
public static final byte MATH_SYMBOL;
public static final byte CURRENCY_SYMBOL;
public static final byte MODIFIER_SYMBOL;
public static final byte OTHER_SYMBOL;
public static final byte INITIAL_QUOTE_PUNCTUATION;
public static final byte FINAL_QUOTE_PUNCTUATION;

@ScopedThis public Character(char v);
@ScopedPure public int compareTo(Character another_character);
@ScopedPure public boolean equals(Object obj);
public static int getType(char ch);
public static int digit(char ch, int radix);
public static boolean isLetter(char ch);
public static boolean isLetterOrDigit(char ch);
public static boolean isLowerCase(char ch);
public static boolean isSpaceChar(char ch);
public static boolean isUpperCase(char ch);
public static boolean isWhitespace(char ch);
public static char toLowerCase(char ch);
public static char toUpperCase(char ch);
@CallerAllocatedResult @ScopedThis public String toString();
@CallerAllocatedResult public static Character valueOf(char c);
}
```

---

---

## Class.java

---

```
package java.lang;

import java.lang.annotation.Annotation;
import java.lang.reflect.*;
import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedPure;
```

```

import javax.realtime.util.sc.ScopedThis;

public class Class<T> {
    public static @Scoped Class<?> forName(String name);
    public @StaticAnalyzable @ScopedThis boolean desiredAssertionStatus();
    public @Scoped @ScopedPure
        <A extends Annotation> A getAnnotation(Class<A> annotationClass);
    public @Scoped @ScopedPure Constructor getConstructor(Class<T>... parameterTypes)
        throws NoSuchMethodException, SecurityException;
    public @ScopedThis @CallerAllocatedResult @ScopedArray Annotation[] getAnnotations();
    public Class<?> getDeclaringClass();
    public @ScopedThis @CallerAllocatedResult @ScopedArray Constructor[] getConstructors();
    public @ScopedThis @CallerAllocatedResult @ScopedArray
        Annotation[] getDeclaredAnnotations();
    public @ScopedThis @CallerAllocatedResult @ScopedArray Field[] getDeclaredFields();
    public @ScopedThis @CallerAllocatedResult @ScopedArray Method[] getDeclaredMethods();
    public @ScopedThis @CallerAllocatedResult T[] getEnumConstants();
    public @ScopedThis @CallerAllocatedResult @ScopedArray Field[] getFields();
    public @ScopedThis @CallerAllocatedResult @ScopedArray Method[] getMethods();
    public @CallerAllocatedResult @ScopedThis String getName();
    public @StaticAnalyzable @ScopedThis @Scoped Class<?> getComponentType();
    public @StaticAnalyzable @ScopedThis boolean isArray();
    public @StaticAnalyzable @ScopedPure boolean isAssignableFrom(Class c);
    public @StaticAnalyzable @ScopedThis boolean isEnum();
    public @StaticAnalyzable @ScopedPure boolean isInstance(Object o);
    public @StaticAnalyzable @ScopedPure boolean isInterface();
    public @StaticAnalyzable @ScopedPure boolean isPrimitive();
    public @ScopedThis T newInstance();
    public @CallerAllocatedResult @ScopedThis String toString();
    public @StaticAnalyzable @ScopedThis boolean isAnnotationType();
}

```

---

### ClassCastException.java

---

```

package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class ClassCastException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis ClassCastException();
    public @StaticAnalyzable @ScopedPure ClassCastException(String msg);
}

```

---

### ClassFormatError.java

---

```

package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;

```

```
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class ClassFormatError extends java.lang.LinkageError {
    public @StaticAnalyzable @ScopedThis ClassFormatError();
    public @StaticAnalyzable @ScopedPure ClassFormatError(String msg);
}
```

---

## ClassLoader.java

---

```
package java.lang;

import javax.realtime.util.Map;
import javax.realtime.util.sc.Scoped;

/**
 * The primordial outer-most parent class loader represents
 * all of the classes loaded at static link time. If a particular run-time
 * implementation includes support for dynamic class loading, that is supported
 * by an inner-nested ClassLoader.
 *
 * To support dynamic class loading and unloading, it is necessary for
 * ClassLoaders to be instantiated within inner-nested ThreadStack contexts. All
 * classes loaded by such a class loader are restricted in visibility to the
 * inner scope. In order to support this abstraction, we need to treat all
 * references to ClassLoader and Class objects as @Scoped.
 */
public class ClassLoader
{
    protected final @Scoped Class< ? > defineClass(String name, byte[] b, int off, int len)
        throws ClassFormatError;

    protected final @Scoped Class< ? > findLoadedClass(String name);

    public final @Scoped ClassLoader getParent();

    /**
     * Within the hard real-time run-time environment, all classes are immediately initialized
     * as soon as they are loaded. This setting applies only to classes loaded after the default
     * setting is modified.
     *
     * @param enabled
     */
    public void setDefaultAssertionStatus(boolean enabled);

    /**
     * This applies only to classes loaded subsequent to setting the class
     * assertion status. To support this capability, the class loader must
     * remember the name and the requested assertion status, so that when/if this
     * class is subsequently loaded, it will have the appropriate assertion
     * status.
     */
}
```

```
*
* @param class_name
* @param enabled
*/
public void setClassAssertionStatus(String class_name, boolean enabled);

/**
 * This applies only to classes loaded subsequent to setting the class
 * assertion status. To support this capability, the class loader must
 * remember the name and the requested assertion status, so that when/if
 * classes from this package are subsequently loaded, the classes will be
 * assigned the appropriate assertion status.
 *
 * @param package_name
 * @param enabled
 */
public void setPackageAssertionStatus(String package_name, boolean enabled);
}
```

---

---

### **ClassNotFoundException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class ClassNotFoundException extends java.lang.Exception {
    public @StaticAnalyzable @ScopedThis ClassNotFoundException();
    public @StaticAnalyzable @ScopedPure ClassNotFoundException(String msg);
}
```

---

---

### **Cloneable.java**

---

```
package java.lang;

public interface Cloneable {
}
```

---

---

### **CloneNotSupportedException.java**

---

```
package java.lang;

public class CloneNotSupportedException extends RuntimeException
{
    public @StaticAnalyzable @ScopedThis CloneNotSupportedException();
    public @StaticAnalyzable @ScopedPure CloneNotSupportedException(String msg);
}
```

---

---

## Comparable.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedPure;

public interface Comparable<T> {
    public @ScopedPure int compareTo(T o) throws ClassCastException;
}
```

---

---

## Double.java

---

```
package java.lang;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.Scoped;

public class Double extends Number implements Comparable<Double>
{
    public static final double MAX_VALUE;
    public static final double MIN_VALUE;
    public static final double NaN;
    public static final double NEGATIVE_INFINITY;
    public static final double POSITIVE_INFINITY;
    public static final Class TYPE;

    @ScopedThis public Double(double val);
    @ScopedPure public Double(String str) throws NumberFormatException;
    @ScopedThis public byte byteValue();
    @ScopedPure public int compareTo(Double other);
    @ScopedThis public double doubleValue();
    @ScopedPure public boolean equals(Object obj);
    public static native long doubleToLongBits(double v);
    public static int compare(double value1, double value2);
    @ScopedThis public float floatValue();
    @ScopedThis public int hashCode();
    @ScopedThis public int intValue();
    @ScopedThis public boolean isInfinite();
    public static boolean isInfinite(double v);
    @ScopedThis public boolean isNaN();
    public static boolean isNaN(double v);
    public static native double longBitsToDouble(long v);
    @ScopedThis public long longValue();
    @ScopedThis public short shortValue();
    @CallerAllocatedResult @StaticAnalyzable @ScopedThis public String toString();
    @CallerAllocatedResult @StaticAnalyzable public static String toString(double v);
    @CallerAllocatedResult @StaticAnalyzable public static Double valueOf(@Scoped String str)
```

```
    throws NumberFormatException;
    @CallerAllocatedResult public static double parseDouble(@Scoped String s)
        throws NumberFormatException;
    public static long doubleToRawLongBits(double val);
}
```

---

---

## Enum.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public abstract class Enum<T> extends Enum<T> implements Comparable<T> {
    protected @ScopedPure Enum(String name, int ordinal);
    public final @StaticAnalyzable @ScopedPure boolean equals(Object o);
    public final @StaticAnalyzable @ScopedThis int hashCode();
    public @StaticAnalyzable @ScopedThis @Scoped String toString();
    public final @ScopedPure int compareTo(T e);
    protected final @CallerAllocatedResult @ScopedThis Object clone();
    public final @StaticAnalyzable @ScopedThis @Scoped String name();
    public @StaticAnalyzable @ScopedThis int ordinal();
    public @StaticAnalyzable @ScopedThis @Scoped final Class<T> getDeclaringClass();
}
```

---

---

## Error.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class Error extends java.lang.Throwable {
    public @StaticAnalyzable @ScopedThis Error();
    public @StaticAnalyzable @ScopedPure Error(String msg);
}
```

---

---

## Exception.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
```

```
public class Exception extends java.lang.Throwable {
    public @StaticAnalyzable @ScopedThis Exception();
    public @StaticAnalyzable @ScopedPure Exception(String msg);
}
```

---

## Float.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.Scoped;

import javax.realtime.util.sc.CallerAllocatedResult;

public class Float extends Number implements Comparable<Float>
{
    public static final Class TYPE;
    public static final float MAX_VALUE;
    public static final float MIN_VALUE;
    public static final float NaN;
    public static final float NEGATIVE_INFINITY;
    public static final float POSITIVE_INFINITY;

    @ScopedThis public Float(float val);
    @ScopedThis public Float(double val);
    @ScopedPure public Float(String str) throws NumberFormatException;
    @ScopedThis public byte byteValue();
    @ScopedPure public int compareTo(Float other);
    @ScopedThis public double doubleValue();
    @ScopedPure public boolean equals(Object obj);
    public static int floatToIntBits(float v);
    public static int compare(float value1, float value2);
    @ScopedThis public float floatValue();
    @ScopedThis public int hashCode();
    public static float intBitsToFloat(int v);
    @ScopedThis public int intValue();
    @ScopedThis public boolean isInfinite();
    public static boolean isInfinite(float v);
    @ScopedThis public boolean isNaN();
    public static boolean isNaN(float v);
    @ScopedThis public long longValue();
    @ScopedThis public short shortValue();
    @CallerAllocatedResult @ScopedThis public String toString();
    @CallerAllocatedResult public static String toString(float v);
    @CallerAllocatedResult public static Float valueOf(@Scoped String str)
        throws NumberFormatException;
    public static float parseFloat(@Scoped String s) throws NumberFormatException;
    public static int floatToRawIntBits(float v);
}
```

---

---

**IllegalArgumentException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class IllegalArgumentException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis IllegalArgumentException();
    public @StaticAnalyzable @ScopedPure IllegalArgumentException(String msg);
}
```

---

---

**IllegalMonitorStateException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class IllegalMonitorStateException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis IllegalMonitorStateException();
    public @StaticAnalyzable @ScopedPure IllegalMonitorStateException(String msg);
}
```

---

---

**IllegalStateException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class IllegalStateException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis IllegalStateException();
    public @StaticAnalyzable @ScopedPure IllegalStateException(String msg);
}
```

---

---

**IndexOutOfBoundsException.java**

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class IndexOutOfBoundsException extends java.lang.RuntimeException {
```

```
    public @StaticAnalyzable @ScopedThis IndexOutOfBoundsException();  
    public @StaticAnalyzable @ScopedPure IndexOutOfBoundsException(String msg);  
}
```

---

## Integer.java

---

```
package java.lang;  
  
import javax.realtime.util.sc.PreallocatedExceptions;  
  
import javax.realtime.util.sc.ScopedThis;  
import javax.realtime.util.sc.ScopedPure;  
import javax.realtime.util.sc.Scoped;  
  
import javax.realtime.util.sc.CallerAllocatedResult;  
  
public class Integer extends Number implements Comparable  
{  
    public static final Class TYPE;  
    public static final int MAX_VALUE;  
    public static final int MIN_VALUE;  
  
    @ScopedThis public Integer(int val);  
    @ScopedPure public Integer(String str) throws NumberFormatException;  
    @ScopedThis public byte byteValue();  
    @ScopedPure public int compareTo(Object o);  
    @ScopedPure public int compareTo(Integer other);  
    @CallerAllocatedResult public static Integer decode(@Scoped String str)  
        throws NumberFormatException;  
    @ScopedThis public double doubleValue();  
    @ScopedPure public boolean equals(Object obj);  
    @ScopedThis public float floatValue();  
    @CallerAllocatedResult public static Integer getInteger(@Scoped String str);  
    @CallerAllocatedResult public static Integer getInteger(@Scoped String str, int v);  
    @CallerAllocatedResult public static Integer getInteger(@Scoped String str, Integer v);  
    @ScopedThis public int hashCode();  
    @ScopedThis public int intValue();  
    @ScopedThis public long longValue();  
    public static int parseInt(@Scoped String str) throws NumberFormatException;  
    public static int parseInt(@Scoped String str, int base) throws NumberFormatException;  
    @ScopedThis public short shortValue();  
    @CallerAllocatedResult public static String toBinaryString(int v);  
    @CallerAllocatedResult public static String toHexString(int v);  
    @CallerAllocatedResult public static String toOctalString(int v);  
    @CallerAllocatedResult @ScopedThis public String toString();  
    @CallerAllocatedResult public static String toString(int v);  
    @CallerAllocatedResult public static String toString(int v, int base);  
    @CallerAllocatedResult public static Integer valueOf(@Scoped String str)  
        throws NumberFormatException;  
    @CallerAllocatedResult public static Integer valueOf(@Scoped String str, int base)  
        throws NumberFormatException;  
}
```

---

---

## InterruptedException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class InterruptedException extends Exception {
    public @StaticAnalyzable @ScopedThis InterruptedException();
    public @StaticAnalyzable @ScopedPure InterruptedException(String s);
}
```

---

---

## Iterable.java

---

```
package java.lang;

import javax.realtime.util.Iterator;

public interface Iterable<T> extends javax.realtime.util.Reconstructable<
{
    Iterator<T> iterator();
}
```

---

---

## LinkageError.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class LinkageError extends java.lang.Error {
    public @StaticAnalyzable @ScopedThis LinkageError();
    public @StaticAnalyzable @ScopedPure LinkageError(String msg);
}
```

---

---

## Long.java

---

```
package java.lang;

import javax.realtime.util.sc.PreallocatedExceptions;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.Scoped;

import javax.realtime.util.sc.CallerAllocatedResult;
```

```

public class Long extends java.lang.Number implements Comparable
{
    public static final long MAX_VALUE;
    public static final long MIN_VALUE;
    public static final Class TYPE;

    @ScopedThis public Long(long val);
    @ScopedPure public Long(String str) throws NumberFormatException;
    @ScopedThis public byte byteValue();
    @ScopedPure public int compareTo(Long other);
    @ScopedPure public int compareTo(Object o);
    @CallerAllocatedResult public static Long decode(@Scoped String str)
        throws NumberFormatException;
    @ScopedThis public double doubleValue();
    @ScopedPure public boolean equals(Object obj);
    @ScopedThis public float floatValue();
    @CallerAllocatedResult public static Long getLong(@Scoped String str);
    @CallerAllocatedResult public static Long getLong(@Scoped String str, long v);
    @CallerAllocatedResult public static Long getLong(@Scoped String str, Long v);
    @ScopedThis public int hashCode();
    @ScopedThis public int intValue();
    @ScopedThis public long longValue();
    public static long parseLong(@Scoped String str) throws NumberFormatException;
    public static long parseLong(@Scoped String str, int base) throws NumberFormatException;
    @ScopedThis public short shortValue();
    @CallerAllocatedResult public static String toBinaryString(long v);
    @CallerAllocatedResult public static String toHexString(long v);
    @CallerAllocatedResult public static String toOctalString(long v);
    @CallerAllocatedResult @ScopedThis public String toString();
    @CallerAllocatedResult public static String toString(long v);
    @CallerAllocatedResult public static String toString(long v, int base);
    @CallerAllocatedResult public static Long valueOf(@Scoped String str)
        throws NumberFormatException;
    @CallerAllocatedResult public static Long valueOf(@Scoped String str, int base)
        throws NumberFormatException;
}

```

---

## Math

---

```

package java.lang;

import java.util.Random;

public strictfp final class Math
{
    @StaticAnalyzable public static final double E;
    @StaticAnalyzable public static final double PI;
    @StaticAnalyzable public static double toRadians(double val);
    @StaticAnalyzable public static double toDegrees(double val);
    @StaticAnalyzable public static double sin(double a);
    @StaticAnalyzable public static double cos(double a);
}

```

```
@StaticAnalyzable public static double tan(double a);
@StaticAnalyzable public static double asin(double a);
@StaticAnalyzable public static double acos(double a);
@StaticAnalyzable public static double atan(double a);
@StaticAnalyzable public static double exp(double a);
@StaticAnalyzable public static double log(double a);
@StaticAnalyzable public static double sqrt(double a);
@StaticAnalyzable public static double IEEERemainder(double f1, double f2);
@StaticAnalyzable public static double ceil(double a);
@StaticAnalyzable public static double floor(double a);
@StaticAnalyzable public static double rint(double a);
@StaticAnalyzable public static double pow(double a, double b);
@StaticAnalyzable private static double atan2_0(double a, double b);
@StaticAnalyzable public static double atan2(double a, double b);
@StaticAnalyzable public static int round(float a);
@StaticAnalyzable public static long round(double a);
@StaticAnalyzable public static double random();
@StaticAnalyzable public static int abs(int a);
@StaticAnalyzable public static long abs(long a);
@StaticAnalyzable public static float abs(float a);
@StaticAnalyzable public static double abs(double a);
@StaticAnalyzable public static int max(int a, int b);
@StaticAnalyzable public static long max(long a, long b);
@StaticAnalyzable public static float max(float a, float b);
@StaticAnalyzable public static double max(double a, double b);
@StaticAnalyzable public static int min(int a, int b);
@StaticAnalyzable public static long min(long a, long b);
@StaticAnalyzable public static float min(float a, float b);
@StaticAnalyzable public static double min(double a, double b);
}
```

---

---

### NegativeArraySizeException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class NegativeArraySizeException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis NegativeArraySizeException();
    public @StaticAnalyzable @ScopedPure NegativeArraySizeException(String msg);
}
```

---

---

### NoSuchMethodException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
```

```
public class NoSuchMethodException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis NoSuchMethodException();
    public @StaticAnalyzable @ScopedPure NoSuchMethodException(String msg);
}
```

---

---

### NullPointerException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class NullPointerException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis NullPointerException();
    public @StaticAnalyzable @ScopedPure NullPointerException(String msg);
}
```

---

---

### Number.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;

public abstract class Number
{
    @ScopedThis public byte byteValue();
    @ScopedThis public abstract double doubleValue();
    @ScopedThis public abstract float floatValue();
    @ScopedThis public abstract long longValue();
    @ScopedThis public abstract int intValue();
    @ScopedThis public short shortValue();
}
```

---

---

### NumberFormatException.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.StaticAnalyzable;

public class NumberFormatException extends IllegalArgumentException
{
    public @StaticAnalyzable @ScopedThis NumberFormatException();
    public @StaticAnalyzable @ScopedPure NumberFormatException(String msg);
}
```

---

---

## Object.java

---

```
package java.lang;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class Object {
    /**
     * The default clone() implementation in standard Java does a shallow (one-level) copy. A
     * difficulty with this approach is that within the hard real-time environment, there are
     * situations (with @CallerAllocatedResult methods or @ReentrantScope object methods)
     * in which a newly constructed object resides in a more outer scope than some of the objects
     * referenced from this object's instance fields. Thus, making the shallow copy would result
     * in IllegalAssignmentExceptions (and the clone method would need to be declared to
     * @AllowCheckedScopedLinks). We want to avoid both of these circumstances.
     *
     * Thus, the byte-code verifier enforces the following:
     *
     * 1. Any class that implements the Cloneable interface but does not provide its
     *    own implementation of the clone() method is not allowed to have any @Scoped
     *    reference instance variables.
     *
     * 2. Any class that implements the Cloneable interface and contains @Scoped reference
     *    instance variables is required to provide its own clone() implementation. Since clone()
     *    is forbidden from having the @AllowCheckedScopedLinks annotation, the
     *    implementor is responsible for cloning the contents of these fields in a safe way.
     *    Presumably, this means the clone() implementation will recursively clone() the
     *    contents of its @Scoped reference instance variables. Note that clone() need not be
     *    @StaticAnalyzable.
     */
    protected @CallerAllocatedResult @ScopedPure Object clone()
        throws CloneNotSupportedException;
    public @ScopedPure boolean equals(Object o);
    public @StaticAnalyzable @ScopedThis @Scoped Class getClass();
    public @StaticAnalyzable @ScopedThis int hashCode();
    public final @StaticAnalyzable @ScopedThis void finalize();
    public @CallerAllocatedResult @ScopedThis String toString();
    public @ScopedThis void notify();
    public @ScopedThis void notifyAll();
    public @ScopedThis void wait();
}
```

---

---

## OutOfMemoryError.java

---

```
package java.lang;
```

```
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class OutOfMemoryError extends java.lang.LinkageError {
    public @StaticAnalyzable @ScopedThis OutOfMemoryError();
    public @StaticAnalyzable @ScopedPure OutOfMemoryError(String msg);
}
```

---

---

## Runnable.java

---

```
package java.lang;

import javax.realtime.util.sc.ScopedThis;

public interface Runnable {
    public @ScopedThis void run();
}
```

---

---

## RuntimeException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class RuntimeException extends java.lang.Exception {
    public @StaticAnalyzable @ScopedThis RuntimeException();
    public @StaticAnalyzable @ScopedPure RuntimeException(String msg);
}
```

---

---

## SecurityException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class SecurityException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis SecurityException();
    public @StaticAnalyzable @ScopedPure SecurityException(String msg);
}
```

---

---

## Short.java

---

```
package java.lang;
```

```
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.Scoped;

import javax.realtime.util.sc.CallerAllocatedResult;

public class Short extends Number implements Comparable<Short>
{
    public static final short MAX_VALUE;
    public static final short MIN_VALUE;
    public static final Class TYPE;

    @ScopedThis public Short(short val);
    @ScopedPure public Short(String str) throws NumberFormatException;
    @ScopedThis public byte byteValue();
    @ScopedPure public int compareTo(Short other);
    @CallerAllocatedResult public static Short decode(@Scoped String str)
        throws NumberFormatException;
    @ScopedThis public double doubleValue();
    @ScopedPure public boolean equals(Object obj);
    @ScopedThis public float floatValue();
    @ScopedThis public int hashCode();
    @ScopedThis public int intValue();
    @ScopedThis public long longValue();
    public static short parseShort(@Scoped String str) throws NumberFormatException;
    public static short parseShort(@Scoped String str, int base)
        throws NumberFormatException;
    @ScopedThis public short shortValue();
    @CallerAllocatedResult @ScopedThis public String toString();
    @CallerAllocatedResult public static String toString(short v);
    @CallerAllocatedResult public static Short valueOf(@Scoped String str)
        throws NumberFormatException;
    @CallerAllocatedResult public static Short valueOf(@Scoped String str, int base)
        throws NumberFormatException;
}
```

---

---

### StackOverflowError.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class StackOverflowError extends java.lang.VirtualMachineError {
    public @StaticAnalyzable @ScopedThis StackOverflowError();
    public @StaticAnalyzable @ScopedPure StackOverflowError(String msg);
}
```

---

---

## StackTraceElement.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class StackTraceElement {
    public @StaticAnalyzable @ScopedThis @Scoped String getClassName();
    public @StaticAnalyzable @ScopedThis @Scoped String getFileName();
    public @StaticAnalyzable @ScopedThis @Scoped String getMethodName();
    public @StaticAnalyzable @ScopedThis int getLineNumber();
}
```

---

---

## String.java

---

```
package java.lang;

import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.CallerAllocatedResult;

import javax.realtime.util.sc.StaticAnalyzable;

public class String
{
    @ScopedThis @ScopedPure public String(byte b[]);
    @ScopedPure public String(byte b[], int offset, int length);
    @ScopedPure public String(char c[]);
    @ScopedPure public String(StringBuilder b);
    @ScopedPure public String(char[] buf, int offset, int length, boolean share_buffer);
    @ScopedPure public String(char c[], int offset, int length);
    @StaticAnalyzable @ScopedThis public char charAt(int index);
    @StaticAnalyzable @ScopedThis public int hashCode();
    @ScopedPure public int compareTo(String str);
    @ScopedPure @CallerAllocatedResult public String concat(String arg);
    @ScopedPure public boolean equals(String arg);
    @ScopedPure public boolean equalsIgnoreCase(String str);
    @ScopedPure public boolean regionMatches(int myoffset, String str, int offset, int len);
    @ScopedPure public boolean
        regionMatches(boolean ignore_case, int myoffset, String str, int offset, int len);
    @ScopedPure public boolean equals(Object obj);
    @ScopedThis @CallerAllocatedResult public byte[] getBytes();
    @ScopedPure public void getChars(int src_begin, int src_end, char dst[], int dst_begin);
    @StaticAnalyzable @ScopedThis public int length();
    @ScopedPure final public boolean startsWith(String prefix);
    @ScopedPure final public boolean endsWith(String suffix);
    @Scoped @ScopedPure @CallerAllocatedResult final public String trim();
}
```

```
@StaticAnalyzable @CallerAllocatedResult @ScopedThis
    public String substring(int begin_index);
@StaticAnalyzable @CallerAllocatedResult @ScopedThis
    public String substring(int begin_index, int end_index);
@ScopedThis @CallerAllocatedResult public char[] toCharArray();
@ScopedThis @CallerAllocatedResult public String toLowerCase();
@ScopedThis @CallerAllocatedResult public String toUpperCase();
public static String valueOf(boolean b);
@CallerAllocatedResult public static String valueOf(char c);
@CallerAllocatedResult public static String valueOf(@Scoped char[] data);
@CallerAllocatedResult public static String valueOf(@Scoped char[] data, int offset, int len);
@CallerAllocatedResult public static String valueOf(double d);
@CallerAllocatedResult public static String valueOf(float f);
@CallerAllocatedResult public static String valueOf(int i);
@CallerAllocatedResult public static String valueOf(long l);
@CallerAllocatedResult public static String valueOf(@Scoped Object o);
}
```

---

---

### StringBuilder.java

---

```
package java.lang;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class StringBuilder {
    public @ScopedThis StringBuilder();
    public @ScopedThis StringBuilder(int length);
    public @ScopedPure StringBuilder(String str);
    public @StaticAnalyzable @ScopedThis StringBuilder append(boolean b);
    public @StaticAnalyzable @ScopedThis StringBuilder append(char c);
    public @StaticAnalyzable @ScopedThis StringBuilder append(double d);
    public @StaticAnalyzable @ScopedThis StringBuilder append(float f);
    public @StaticAnalyzable @ScopedThis StringBuilder append(int i);
    public @StaticAnalyzable @ScopedThis StringBuilder append(long l);
    public @ScopedThis StringBuilder append(@Scoped Object o);
    public @ScopedThis StringBuilder append(@Scoped String s);
    public @ScopedThis StringBuilder append(@Scoped StringBuilder sb);
    public @StaticAnalyzable @ScopedThis int length();
    public @StaticAnalyzable @ScopedThis int capacity();
    public @CallerAllocatedResult @ScopedThis String toString();
    private @StaticAnalyzable @CallerAllocatedResult @ScopedThis String toStringNoCopy();
}
```

---

---

### StringIndexOutOfBoundsException.java

---

```
package java.lang;
```

```
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

import javax.realtime.util.sc.StaticAnalyzable;

public class StringIndexOutOfBoundsException extends IndexOutOfBoundsException
{
    public @StaticAnalyzable @ScopedThis StringIndexOutOfBoundsException();
    public @StaticAnalyzable @ScopedPure StringIndexOutOfBoundsException(String msg);
}
```

---

---

## Thread.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public class Thread implements Runnable {
    public @ScopedThis Thread();
    public @ScopedThis Thread(String name);
    public @Scoped static Thread currentThread();
    public static void sleep(long millis);
    public static void sleep(long millis, int nanos);
    public static @StaticAnalyzable @ScopedThis void yield();
    public @StaticAnalyzable @ScopedThis @Scoped String getName();
    public @StaticAnalyzable @ScopedThis boolean isAlive();
    public @StaticAnalyzable @ScopedThis boolean isDaemon();
    public @ScopedThis void join();
    public @ScopedThis void join(long millis);
    public @ScopedThis void join(long millis, int nanos);
    public @ScopedThis void run();
    public @ScopedThis void start();
}
```

---

---

## Throwable.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.CallerAllocatedArrayResult;
import javax.realtime.util.sc.CallerAllocatedResult;

public class Throwable {
    public @StaticAnalyzable @ScopedThis Throwable();
    public @StaticAnalyzable @ScopedPure Throwable(String msg);
}
```

```
public @StaticAnalyzable @ScopedThis @Scoped String getMessage();
public @StaticAnalyzable @CallerAllocatedArrayResult @ScopedThis
    StackTraceElement[] getStackTrace() throws IllegalStateException;
}
```

---

---

### UndeclaredThrowableException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class UndeclaredThrowableException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis UndeclaredThrowableException();
    public @StaticAnalyzable @ScopedPure UndeclaredThrowableException(String msg);
}
```

---

---

### UnsupportedOperationException.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.CallerAllocatedArrayResult;
import javax.realtime.util.sc.CallerAllocatedResult;

public class UnsupportedOperationException extends RuntimeException {
    public @StaticAnalyzable @ScopedThis UnsupportedOperationException();
    public @StaticAnalyzable @ScopedPure UnsupportedOperationException(String msg);
}
```

---

---

### VirtualMachineError.java

---

```
package java.lang;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class VirtualMachineError extends java.lang.Error {
    public @StaticAnalyzable @ScopedThis VirtualMachineError();
    public @StaticAnalyzable @ScopedPure VirtualMachineError(String msg);
}
```

## 2. Subset of the JDK 1.5 `java.lang.annotation` Package

---

---

### **Annotation.java**

---

```
package java.lang.annotation;

public interface Annotation {
    public abstract boolean equals(java.lang.Object o);
    public abstract int hashCode();
    public abstract java.lang.String toString();
    public abstract java.lang.Class annotationType();
}
```

---

---

### **AnnotationInfo.java**

---

```
package java.lang.annotation;

public class AnnotationInfo {
    public static Annotation[] getAnnotations(String element_name);
    Annotation[][] getParameterAnnotations(String element);
}
```

---

---

### **Documented.java**

---

```
package java.lang.annotation;

@Documented @Target(ElementType.ANNOTATION_TYPE) public @interface Documented {
}
```

---

---

### **ElementType.java**

---

```
package java.lang.annotation;

public enum ElementType {
    TYPE,
    FIELD,
    METHOD,
    PARAMETER,
    CONSTRUCTOR,
    LOCAL_VARIABLE,
    ANNOTATION_TYPE,
    PACKAGE
}
```

---

---

### **Inherited.java**

---

```
package java.lang.annotation;
```

```
@Documented @Retention(RetentionPolicy.RUNTIME)
  @Target(ElementType.ANNOTATION_TYPE) public @interface Inherited {
}
```

---

---

### **Overrides.java**

---

```
package java.lang.annotation;

@Documented @Retention(RetentionPolicy.SOURCE) @Target(ElementType.METHOD)
public @interface Overrides {
}
```

---

---

### **Retention.java**

---

```
package java.lang.annotation;

@Documented @Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE) public @interface Retention {
  RetentionPolicy value();
}
```

---

---

### **RetentionPolicy.java**

---

```
package java.lang.annotation;

public enum RetentionPolicy {
  SOURCE, CLASS, RUNTIME
}
```

---

---

### **Target.java**

---

```
package java.lang.annotation;

@Documented @Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE) public @interface Target {
  ElementType[] value();
}
```

## **3. Subset of the JDK 1.5 java.lang.reflect Package**

---

---

### **AccessibleObject.java**

---

```
package java.lang.reflect;

import java.lang.annotation.Annotation;
import java.lang.reflect.AnnotatedElement;
```

(Permission granted to reproduce and distribute as a complete document, without modification)

```
import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class AccessibleObject implements AnnotatedElement {
    public @ScopedPure @Scoped
        <T extends Annotation> T getAnnotation(Class<T> annotationClass);
    public @ScopedPure boolean isAnnotationPresent(java.lang.Class<?
        extends java.lang.annotation.Annotation> c);
    public @ScopedThis @CallerAllocatedResult @ScopedArray Annotation[] getAnnotations();
    public @ScopedThis @CallerAllocatedResult @ScopedArray
        Annotation[] getDeclaredAnnotations();
}
```

---

---

### **AnnotatedElement.java**

---

```
package java.lang.reflect;

import java.lang.annotation.Annotation;
import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public interface AnnotatedElement {
    public @ScopedPure @Scoped
        <T extends Annotation> T getAnnotation(Class<T> annotationType);
    public @ScopedPure Annotation[] getAnnotations();
    public @ScopedThis @CallerAllocatedResult @ScopedArray
        Annotation[] getDeclaredAnnotations();
    public @ScopedPure @CallerAllocatedResult @ScopedArray
        boolean isAnnotationPresent(java.lang.Class<? extends java.lang.annotation.Annotation> c);
}
```

---

---

### **Constructor.java**

---

```
package java.lang.reflect;

import java.lang.annotation.Annotation;
import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
```

```
public final class Constructor extends AccessibleObject implements Member {
    public @StaticAnalyzable @ScopedThis @Scoped String getName();
    public @ScopedPure @Scoped
        <T extends Annotation> T getAnnotation(Class<T> annotationClass);
    public @ScopedThis @CallerAllocatedResult @ScopedArray
        Annotation[] getDeclaredAnnotations();
    public @StaticAnalyzable @ScopedThis boolean isSynthetic();
}
```

---

---

## Field.java

---

```
package java.lang.reflect;

import java.lang.annotation.Annotation;
import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public final class Field extends AccessibleObject implements Member {
    public @StaticAnalyzable @ScopedThis @Scoped String getName();
    public @ScopedPure @Scoped
        <T extends Annotation> T getAnnotation(Class<T> annotationClass);
    public @ScopedThis @CallerAllocatedResult @ScopedArray
        Annotation[] getDeclaredAnnotations();
    public @StaticAnalyzable @ScopedThis boolean isSynthetic();
}
```

---

---

## Member.java

---

```
package java.lang.reflect;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public interface Member {
    @StaticAnalyzable @ScopedThis @Scoped String getName();
    @StaticAnalyzable @ScopedThis boolean isSynthetic();
}
```

---

---

## Method.java

---

```
package java.lang.reflect;

import javax.realtime.util.sc.CallerAllocatedResult;
```

```
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedArray;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import java.lang.annotation.Annotation;

public final class Method extends java.lang.reflect.AccessibleObject implements Member{
    public @StaticAnalyzable @Scoped @ScopedThis String getName();
    public @Scoped @ScopedPure
        <T extends Annotation> T getAnnotation(Class<T> annotationClass);
    public @CallerAllocatedResult @ScopedThis @ScopedArray
        Annotation[] getDeclaredAnnotations();
    public @StaticAnalyzable @ScopedThis boolean isSynthetic();
}
```

#### 4. Subset of the JDK 1.5 java.io Package

---

---

##### Serializable

---

```
package java.io;

public interface Serializable
{
}
```

#### 5. Subset of the JDK 1.5 java.util Package

---

---

##### Random

---

```
package java.util;

import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class Random implements java.io.Serializable
{
    @StaticAnalyzable @ScopedThis public Random();
    @StaticAnalyzable @ScopedThis public Random(long seed);
    @StaticAnalyzable @ScopedThis public synchronized void setSeed(long seed);
    @StaticAnalyzable @ScopedThis protected synchronized int next(int bits);
    @StaticAnalyzable @ScopedPure public void nextBytes(byte bytes[]);
    @StaticAnalyzable @ScopedThis public int nextInt();
    @StaticAnalyzable @ScopedThis public int nextInt(int n);
    @StaticAnalyzable @ScopedThis public long nextLong();
    @StaticAnalyzable @ScopedThis public boolean nextBoolean();
    @StaticAnalyzable @ScopedThis public float nextFloat();
    @StaticAnalyzable @ScopedThis public double nextDouble();
    @StaticAnalyzable @ScopedThis public synchronized double nextGaussian();
}
```

## Appendix B: Hard Real-Time Subset of Real-Time Specification for Java

Note that both the number of classes and the number of methods supported by each class have been reduced significantly from the official Real-Time Specification for Java standard. To reduce space and formatting effort, this subset description is provided only in terms of raw Java type declarations. The semantic descriptions of the supported classes and methods are available in standard Java Community Process documentation.

Note also that, though they are present, hard real-time developers should avoid using the following classes:

- `AbsoluteTime`
- `AperiodicParameters`
- `AsyncEvent`
- `BoundAsyncEventHandler`
- `Clock`
- `HighResolutionTime`
- `NoHeapRealtimeThread`
- `OneShotTimer`
- `PeriodicParameters`
- `PeriodicTimer`
- `RelativeTime`
- `ReleaseParameters`
- `SizeEstimator`
- `SporadicParameters`
- `Timer`

In place of these classes, use the replacement classes by the same name in the `javax.realtime.util.sc` package. Each replacement class extends the `javax.realtime` class, so the replacements can serve in any context that would be appropriate for the original.

---

---

### **AbsoluteTime.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class AbsoluteTime extends HighResolutionTime {
    public @StaticAnalyzable @ScopedPure AbsoluteTime(AbsoluteTime time)
        throws IllegalArgumentException;
    public @StaticAnalyzable @ScopedPure AbsoluteTime(AbsoluteTime time, Clock clock)
        throws IllegalArgumentException;
    public @StaticAnalyzable @ScopedThis
        AbsoluteTime(long millis, int nanos) throws IllegalArgumentException;
```

(Permission granted to reproduce and distribute as a complete document, without modification)

```
public @StaticAnalyzable @ScopedThis AbsoluteTime(long millis, int nanos, Clock clock)
    throws IllegalArgumentException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis AbsoluteTime absolute();
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    AbsoluteTime add(long millis, int nanos) throws ArithmeticException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedPure
    AbsoluteTime add(RelativeTime time)
    throws IllegalArgumentException, ArithmeticException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedPure
    RelativeTime subtract(AbsoluteTime time)
    throws IllegalArgumentException, ArithmeticException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedPure
    AbsoluteTime subtract(RelativeTime time)
    throws IllegalArgumentException, ArithmeticException;
public @StaticAnalyzable @ScopedThis @CallerAllocatedResult String toString();
}
```

---

---

### AperiodicParameters.java

---

```
package javax.realtime;

public class AperiodicParameters extends ReleaseParameters {
}
```

---

---

### ArrivalTimeQueueOverflowException.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class ArrivalTimeQueueOverflowException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis ArrivalTimeQueueOverflowException();
    public @StaticAnalyzable @ScopedPure
        ArrivalTimeQueueOverflowException(String description);
}
```

---

---

### AsyncEvent.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
```

```
public class AsyncEvent {
    public @StaticAnalyzable @ScopedThis void fire()
        throws ArrivalTimeQueueOverflowException, MITViolationException;
    public @StaticAnalyzable @ScopedPure boolean handledBy(AsyncEventHandler handler);
}
```

---

---

### AsyncEventHandler.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class AsyncEventHandler implements Schedulable {
    public static @StaticAnalyzable int numProxyServers(int priority)
        throws IllegalArgumentException;
    public static @StaticAnalyzable int numActiveProxies(int priority)
        throws IllegalArgumentException;
    public static @StaticAnalyzable int numBlockedProxies(int priority)
        throws IllegalArgumentException;
    public static void setProxyServers(int priority, int num_proxies)
        throws IllegalArgumentException, IllegalStateException, OutOfMemoryError;
    public @StaticAnalyzable AsyncEventHandler(SchedulingParameters scheduling,
        ReleaseParameters release, MemoryParameters memory,
        boolean is_daemon) throws IllegalArgumentException;
    public @StaticAnalyzable int getAndClearPendingFireCount();
    public @StaticAnalyzable int getAndDecrementPendingFireCount();
    public @StaticAnalyzable MemoryArea getMemoryArea();
    public @StaticAnalyzable MemoryParameters getMemoryParameters();
    protected @StaticAnalyzable int getPendingFireCount();
    public @StaticAnalyzable ReleaseParameters getReleaseParameters();
    public @StaticAnalyzable SchedulingParameters getSchedulingParameters();
    public @StaticAnalyzable int getOverflowTriggers();
    public @StaticAnalyzable int getViolationTriggers();
    public void handleAsyncEvent();
    public final void run();
    public @StaticAnalyzable final boolean isDaemon();
}
```

---

---

### BoundAsyncEventHandler.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;

public class BoundAsyncEventHandler extends AsyncEventHandler {
```

```
protected @StaticAnalyzable @ScopedPure
    BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
        MemoryParameters memory, boolean is_daemon)
    throws IllegalArgumentException;
}
```

---

---

## Clock.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class Clock {
    public @StaticAnalyzable @CallerAllocatedResult RelativeTime getEpochOffset();
    public @StaticAnalyzable @CallerAllocatedResult RelativeTime getResolution();
    public @StaticAnalyzable @CallerAllocatedResult AbsoluteTime getTime();
    public @StaticAnalyzable static Clock getRealtimeClock();
}
```

---

---

## HighResolutionTime.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public abstract class HighResolutionTime implements java.lang.Comparable {
    public @StaticAnalyzable @ScopedPure int compareTo(HighResolutionTime time)
        throws ClassCastException, IllegalArgumentException;
    public @StaticAnalyzable @ScopedPure int compareTo(Object object)
        throws ClassCastException, IllegalArgumentException;
    public @StaticAnalyzable @ScopedPure boolean equals(HighResolutionTime time);
    public @StaticAnalyzable @ScopedPure boolean equals(Object object);
    public @StaticAnalyzable @ScopedThis Clock getClock();
    public @StaticAnalyzable @ScopedThis final long getMilliseconds();
    public @StaticAnalyzable @ScopedThis final int getNanoseconds();
}
```

---

---

## ImmortalMemory.java

---

```
package javax.realtime;

import javax.realtime.util.sc.Configuration;
```

```
import javax.realtime.util.sc.StaticAnalyzable;

public final class ImmortalMemory extends MemoryArea {
    private ImmortalMemory();
    public static @StaticAnalyzable ImmortalMemory instance();
}
```

---

---

### **LTMemory.java**

---

```
package javax.realtime;

public class LTMemory extends ScopedMemory {
    public LTMemory();
}
```

---

---

### **MITViolationException.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class MITViolationException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis MITViolationException();
    public @StaticAnalyzable @ScopedPure MITViolationException(String description);
}
```

---

---

### **MemoryArea.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public abstract class MemoryArea {
    protected MemoryArea();
    protected MemoryArea(long size) throws IllegalArgumentException, OutOfMemoryError;
    protected MemoryArea(long size, Runnable logic)
        throws IllegalArgumentException, OutOfMemoryError;
    protected MemoryArea(SizeEstimator size)
        throws IllegalArgumentException, OutOfMemoryError;
    public @StaticAnalyzable long memoryConsumed();
    public @StaticAnalyzable long memoryRemaining();
}
```

```
    public @StaticAnalyzable long size();
    public MemoryArea(long initial, long maximum, java.lang.Runnable logic);
    public void join() throws InterruptedException;
}
```

---

---

### MemoryParameters.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class MemoryParameters implements Cloneable {
    public static final long NO_MAX = -1L;
    public @StaticAnalyzable MemoryParameters(long max_memory_area, long max_immortal)
        throws IllegalArgumentException;
    public @StaticAnalyzable long getMaxImmortal();
}
```

---

---

### MonitorControl.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;

public abstract class MonitorControl {
    protected @StaticAnalyzable MonitorControl();
    public static @StaticAnalyzable MonitorControl getMonitorControl();
    public static @StaticAnalyzable MonitorControl getMonitorControl(Object obj);
    public static @StaticAnalyzable MonitorControl setMonitorControl(MonitorControl policy);
}
```

---

---

### NoHeapRealtimeThread.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.Scoped;

public abstract class NoHeapRealtimeThread extends RealtimeThread {
    public @StaticAnalyzable @ScopedPure
        NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)
```

```
    throws java.lang.IllegalArgumentException;
public @StaticAnalyzable @ScopedPure
    NoHeapRealtimeThread(SchedulingParameters scheduling, ReleaseParameters release,
        MemoryArea area)
    throws java.lang.IllegalArgumentException;
}
```

---

---

### OneShotTimer.java

---

```
package javax.realtime;

public class OneShotTimer extends Timer {
    public OneShotTimer(HighResolutionTime time, AsyncEventHandler handler)
        throws IllegalArgumentException;
}
```

---

---

### PeriodicParameters.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class PeriodicParameters extends ReleaseParameters {
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis RelativeTime getPeriod();
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
        HighResolutionTime getStart();
}
```

---

---

### PeriodicTimer.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class PeriodicTimer extends Timer {
    public @StaticAnalyzable PeriodicTimer(HighResolutionTime start, RelativeTime interval,
        AsyncEventHandler handler) throws IllegalArgumentException;
    public @StaticAnalyzable @CallerAllocatedResult RelativeTime getInterval();
    public @StaticAnalyzable @CallerAllocatedResult AbsoluteTime getFireTime();
}
```

---

---

### PriorityCeilingEmulation.java

---

```
package javax.realtime;
```

```
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class PriorityCeilingEmulation extends MonitorControl {
    private @StaticAnalyzable PriorityCeilingEmulation();
    public @StaticAnalyzable int getCeiling();
    public static @StaticAnalyzable PriorityCeilingEmulation getMaxCeiling();
    public static @StaticAnalyzable PriorityCeilingEmulation
        instance(int ceiling) throws IllegalArgumentException;
}
```

---

---

### PriorityInheritance.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class PriorityInheritance extends MonitorControl {
    private @StaticAnalyzable PriorityInheritance();
    public static PriorityInheritance instance();
}
```

---

---

### PriorityParameters.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class PriorityParameters extends SchedulingParameters {
    public @StaticAnalyzable @ScopedThis PriorityParameters(int priority)
        throws IllegalArgumentException;
    public @StaticAnalyzable @ScopedThis int getPriority();
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis String toString();
}
```

---

---

### PriorityScheduler.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;
```

```
public class PriorityScheduler extends Scheduler {
    public static @StaticAnalyzable int getMaxPriority();
    public static @StaticAnalyzable int getMinPriority();
    public static @StaticAnalyzable PriorityScheduler instance();
}
```

---

---

## RealtimeThread.java

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.Scoped;

public abstract class RealtimeThread extends Thread implements Schedulable {
    protected @StaticAnalyzable @ScopedPure RealtimeThread();
    protected @StaticAnalyzable @ScopedPure
        RealtimeThread(SchedulingParameters scheduling);
    protected @StaticAnalyzable @ScopedPure
        RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release);
    protected @StaticAnalyzable @ScopedPure
        RealtimeThread(String name, SchedulingParameters scheduling, ReleaseParameters release,
            MemoryParameters memory, MemoryArea area);
    public static @StaticAnalyzable @Scoped RealtimeThread currentRealtimeThread()
        throws ClassCastException;
    public @StaticAnalyzable @Scoped @ScopedThis
        MemoryParameters getMemoryParameters();
    public @StaticAnalyzable @Scoped @ScopedThis
        ReleaseParameters getReleaseParameters();
    public @StaticAnalyzable @Scoped @ScopedThis
        SchedulingParameters getSchedulingParameters();
    public static @StaticAnalyzable @ScopedThis void sleep(@Scoped HighResolutionTime time)
        throws IllegalArgumentException;
    public @StaticAnalyzable @ScopedThis void start();
}
```

---

---

## RelativeTime.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class RelativeTime extends HighResolutionTime {
    public @StaticAnalyzable @ScopedThis RelativeTime(long millis, int nanos)
        throws ArithmeticException;
    public @StaticAnalyzable @ScopedThis RelativeTime(long millis, int nanos, Clock clock)
        throws ArithmeticException;
}
```

```
public @StaticAnalyzable @ScopedPure RelativeTime(RelativeTime time, Clock clock);
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis AbsoluteTime absolute();
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    RelativeTime add(long millis, int nanos) throws ArithmeticException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedPure
    RelativeTime add(RelativeTime time)
        throws ArithmeticException, IllegalArgumentException;
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis RelativeTime relative();
public @StaticAnalyzable @CallerAllocatedResult @ScopedPure
    RelativeTime subtract(RelativeTime time)
        throws IllegalArgumentException, ArithmeticException;
}
```

---

---

### ReleaseParameters.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class ReleaseParameters {
    public static final String arrivalTimeQueueOverflowIgnore = "ignore";
    public static final String arrivalTimeQueueOverflowReplace = "replace";

    protected ReleaseParameters();
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis RelativeTime getDeadline();
    public @StaticAnalyzable @Scoped @ScopedThis
        AsyncEventHandler getDeadlineMissHandler();
}
```

---

---

### Schedulable.java

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public interface Schedulable extends java.lang Runnable {
    public @StaticAnalyzable @Scoped @ScopedThis
        ReleaseParameters getReleaseParameters();
    public @StaticAnalyzable @Scoped @ScopedThis
        MemoryParameters getMemoryParameters();
    public @StaticAnalyzable @Scoped @ScopedThis
        SchedulingParameters getSchedulingParameters();
}
```

---

---

## **Scheduler.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public abstract class Scheduler {
    protected @StaticAnalyzable Scheduler();
}
```

---

---

## **SchedulingParameters.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedThis;

public abstract class SchedulingParameters {
    public @StaticAnalyzable @ScopedThis SchedulingParameters();
}
```

---

---

## **ScopedMemory.java**

---

```
package javax.realtime;

public class ScopedMemory extends MemoryArea {
}
```

---

---

## **SizeEstimator.java**

---

```
package javax.realtime;

import javax.realtime.util.sc.CallerAllocatedResult;
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;

public class SizeEstimator {
    public @StaticAnalyzable @ScopedThis SizeEstimator();
    public @StaticAnalyzable @ScopedThis long getEstimate();
    public @StaticAnalyzable @ScopedPure void reserve(Class c, int number);
    public @StaticAnalyzable @ScopedPure void reserve(SizeEstimator size);
    public @StaticAnalyzable @ScopedPure
        void reserve(@Scoped SizeEstimator estimator, int number);
    public @StaticAnalyzable @ScopedThis void reserveArray(int dimension);
}
```

```
    public @StaticAnalyzable @ScopedPure void reserveArray(int dimension, Class type);  
}
```

---

---

### SporadicParameters.java

---

```
package javax.realtime;  
  
import javax.realtime.util.sc.StaticAnalyzable;  
import javax.realtime.util.sc.ScopedThis;  
import javax.realtime.util.sc.CallerAllocatedResult;  
  
public class SporadicParameters extends AperiodicParameters {  
    protected SporadicParameters();  
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis  
        RelativeTime getMinimumInterarrival();  
    public @StaticAnalyzable @ScopedThis AsyncEventHandler getMitViolationHandler();  
}
```

---

---

### Timer.java

---

```
package javax.realtime;  
  
import javax.realtime.util.sc.CallerAllocatedResult;  
import javax.realtime.util.sc.StaticAnalyzable;  
import javax.realtime.util.sc.ScopedPure;  
import javax.realtime.util.sc.ScopedThis;  
import javax.realtime.util.sc.Scoped;  
  
public class Timer extends AsyncEvent {  
    protected Timer();  
    public @StaticAnalyzable @ScopedThis void destroy() throws IllegalStateException;  
    public @StaticAnalyzable @ScopedThis void disable() throws IllegalStateException;  
    public @StaticAnalyzable @ScopedThis void enable() throws IllegalStateException;  
    public final @StaticAnalyzable @ScopedThis void fire()  
        throws UnsupportedOperationException;  
    public @StaticAnalyzable @ScopedThis Clock getClock();  
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis  
        AbsoluteTime getFireTime() throws IllegalStateException;  
    public @StaticAnalyzable @ScopedThis boolean isRunning() throws IllegalStateException;  
    public @StaticAnalyzable @ScopedPure void reschedule(HighResolutionTime time)  
        throws IllegalStateException;  
    public @ScopedThis void start() throws IllegalStateException;  
    public @ScopedThis void start(boolean disabled) throws IllegalStateException;  
    public @StaticAnalyzable @ScopedThis void stop() throws IllegalStateException;  
}
```

---

---

### UnknownHappeningException.java

---

```
package javax.realtime;
```

```
import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.CallerAllocatedResult;

public class UnknownHappeningException extends java.lang.RuntimeException {
    public @StaticAnalyzable @ScopedThis UnknownHappeningException();
    public @StaticAnalyzable @ScopedPure UnknownHappeningException(String description);
}
```

## Appendix C: Hard Real-Time Extensions to Real-Time Specification for Java

### 1. Annotations to Support Consistent Initialization of Class Variables

In traditional Java, class variables are to be initialized “immediately before first use”. This requires a run-time check and hinders efficient translation of programs for native execution. Further, it introduces certain race conditions in which the initial values of particular class variables (even the values of certain `final` variables) depend on the sequence in which classes are accessed (and initialized).

For the safety-critical Java standard, the ideal is to initialize all class variables prior to run time. A smart linker performs a dependency analysis on all class variables and initializes variables according to a topological sort of the dependency chain. In the best of cases, all class variables are initialized in the static load image prepared by the intelligent linker. When this is not possible, certain class variables will be “scheduled” by the smart linker to be initialized during system startup, prior to execution of any Java threads. In all cases, the order of the initialization sequence is fully determined by the static linker.

Any dependency cycles will be identified at link time. Certain restrictions must be imposed on developers who are writing initialization expressions for class variables in order to facilitate this semantics.

The following restrictions are recommended. These guidelines should be enforced by an appropriate byte-code verifier.

1. Each static (class) variable must be initialized exactly once, outside of any looping or conditional control structures, either with an assignment that is part of the variable’s declaration or with an assignment statement within the body of a static initializer block.
2. In either case, all initialization code is restricted according to the following rules:
  - a. The code is not allowed to perform any side effects (no synchronization, no I/O, no increment or decrement operations, and no assignments) except for the following:
    - i. A single assignment appearing outside of any looping control structures must be provided for each class variable.
    - ii. Assignments are allowed to the instance fields of objects that were located in the current evaluation context and which are not reachable from any class variable (i.e. temporary objects).
    - iii. Assignments are allowed to the local variables of a static initialization context.
  - b. Only `final` methods may be invoked from within initialization code.
3. Since all methods invoked from within initialization contexts are declared `final`, it is possible to generate special “translations” of this code. Within an initialization context, all assignments to class and instance variables are treated as if the variables had been declared `volatile`. This means no code that syntactically precedes the assignment may be reordered to be executed following the assignment. The most straightforward implementation of initialization uses a simple byte-code interpreter. There is little need to accelerate the initialization code.
4. For each declared class, the compiler separates the class variable initialization declarations into one expression for each variable to be initialized. The linker analyzes the byte code of the assignment expression to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies).

**(Permission granted to reproduce and distribute as a complete document, without modification)**

5. For each static class initialization statement, the compiler analyzes the body of the class initialization statement to determine which class variables it initializes and which class variables it depends on. The static class initialization statement is treated as an indivisible unit. This block of code will be scheduled for execution after all dependencies have been resolved. Once executed, any class variables that are defined by this initialization statement will likewise be treated as resolved.
6. When analyzing dependencies, the linker makes no attempt to analyze the dependencies represented by native methods occurring in the initialization expressions. Programmers may supply special annotations to force an ordering on execution of these initialization blocks. The following annotation is supported:

```
class Foo {  
  
    @StaticDependency(Foo.class, "global_C")  
    public static int global_A = nativeMethod1();  
  
    @StaticDependency(Foo.class, "global_A")  
    public static int global_B = nativeMethod2();  
  
    public static int global_C = nativeMethod3();  
  
}
```

These annotations require that the initialization expression for `Foo.global_C (nativeMethod3())` be executed before the initialization expression for `Foo.global_A (nativeMethod1())`, which itself must be executed before the initialization expression for `Foo.global_B (nativeMethod2())`.

7. Because most hard real-time development will involve the use of cross-development tools, it will not be possible in general for a static linker to execute the native code at link time. However, there may be development environments that are able to emulate the deployment platform, and thus might try to run native initialization code in the emulated environment rather than waiting until the system starts up. In the case that a developer knows it would not generally be appropriate to execute certain class initialization (either native code or Java code) prior to run time, the developer may annotate certain variables using the following annotation:

```
@InitializeAtStartup
```

This annotation indicates that the initialization code associated with the class variable that immediately follows must not be executed until the deployed system begins to execute. In case initialization of other variables depend on the initialization of this variable, execution of the initialization code for those other variables must also be delayed until startup time.

8. For each static initializer block, the linker examines the byte code of the static initializer to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies) and which class variables are initialized by execution of this static initializer block. If this block does not initialize any class variables, the block is considered dead and a warning is issued by the compiler.

Consider, for example, the following class declarations:

```
[1] class Foo {  
[2]     static int x = 5;  
[3]     static float z = x + (new Baz()).code1();  
[4]     static Baz b = new Baz(z);
```

**(Permission granted to reproduce and distribute as a complete document, without modification)**

```
[5] static double sines[];
[6] static double cosines[];
[7]
[8]                                     // Source adapted from Flanagan, "Java in a Nutshell", 1996
[9] static {
[10]     double x, delta_x;
[11]     double local_sines[] = new double[1000];
[12]     double local_cosines[] = new double[1000];
[13]     int i;
[14]     delta_x = (Math.PI/2)/(1000-1);
[15]
[16]     // assignments to local_sines[i] and local_cosines[i] are allowed because these
[17]     // local variables refer to locally allocated objects that are not yet visible to
[18]                                     // any class variables.
[19]     for (i = 0; x = 0.0; i < 1000; i++, x+= delta_x) {
[20]         local_sines[i] = Math.sin(x);
[21]         local_cosines[i] = Math.cos(x);
[22]     }
[23]
[24]     sines = local_sines;
[25]     cosines = local_cosines;
[26]
[27]     // no further modification of the local_sines or local_cosines is permitted
[28]     // because these objects are now visible to "global" class variables
[29] }
[30]
[31]     etc ...
[32] }
[33]
[34] class Baz {
[35]     static float y = (float) Foo.x + Foo.b.code2();
[36]     float my_float;
[37]
[38]     Baz() {
[39]         my_float = Math.PI;
[40]     }
[41]
[42]     final float code1() {
[43]         return my_float;
[44]     }
[45]
[46]     final float code2() {
[47]         return (float) 0.693147;
[48]     }
[49] }
[50]
[51] class Math {
[52]     static double PI = 3.141519;
[53]
[54]     static final native double cos(double x);
[55]     static final native double sin(double x);
[56]
[57]     etc ...
```

```
[58] }  
[59]
```

This results in the creation of 6 entries in the initialization dependency graph:

1. `Foo.x = 5`: no dependencies
2. `Foo.b = new Baz(z)`: depends on `Foo.z`.
3. `Foo.z = x + (new Baz()).code1()`: depends on `Foo.x`, `Math.PI`.
4. `Foo.sines` and `Foo.cosines` are initialized by the static initialization code, lines 9 through 29 inclusive, which depends on `Math.PI`. Note that execution of this static initialization code depends on the execution of two native methods. Assume, for the purposes of illustration, that the development tools do not have the ability to emulate execution of these native methods at static link time. Thus, we will defer execution of this particular initialization code until run time, even though there is no `@InitializeAtStartup` annotation that would require it.
5. `Baz.y = (float) Foo.x + Foo.b.code2()`: depends on `Foo.x` and `Foo.b`.
6. `Math.PI = 3.141519`: no dependencies.

Based on the analysis described above, the hard real-time Java linker builds a dependency graph that relates the various executable blocks together, as shown in Figure 3. The initialization evaluator implements the following algorithm:

1. The `Executable_Initializations` list represents all of the expressions for which all dependencies have already been resolved (i.e. the value of the `dependencies` field equals zero). In Figure 3, the expressions on this list are linked through the field illustrated in the top right corner of each expression node. Initially, only the two nodes for which `dependencies` equals zero are on this list.
2. Initialization proceeds by removing the leading entry from the `Executable_Initializations` list.
  - a. If the corresponding initialization expression can be executed prior to system startup, execute the code associated with that entry, and then decrement the `dependencies` count for each of the `dependants` of the evaluated expression. If the new value of the `dependencies` count after performing the decrement operation equals zero, add this node to the `Executable_Initializations` list.
  - b. Otherwise (this initialization code cannot be executed prior to system startup), place this entry on a different list, named the `Startup_Initializations` list.
3. Continue this process until the `Executable_Initializations` list is empty.
4. Now, remove the leading entry from the `Startup_Initializations` list. Schedule the corresponding initialization code to be executed “next” in the startup sequence. Then decrement the `dependencies` count for each of the `dependants` of the evaluated expression. If the new value of the `dependencies` count after performing the decrement operation equals zero, add this node to the `Startup_Initializations` list.
5. Continue this process until the `Startup_Initializations` list is empty.
6. Upon emptying the `Executable_Initializations` and `Startup_Initializations` lists, ask if all expressions have been evaluated. (Keep a count of the total number of expressions to be evaluated and decrement this count each time an expression is evaluated.) If unevaluated expressions remain to be processed, the original program had a dependency cycle in it. The linker issues an error message. The static load image cannot be constructed.

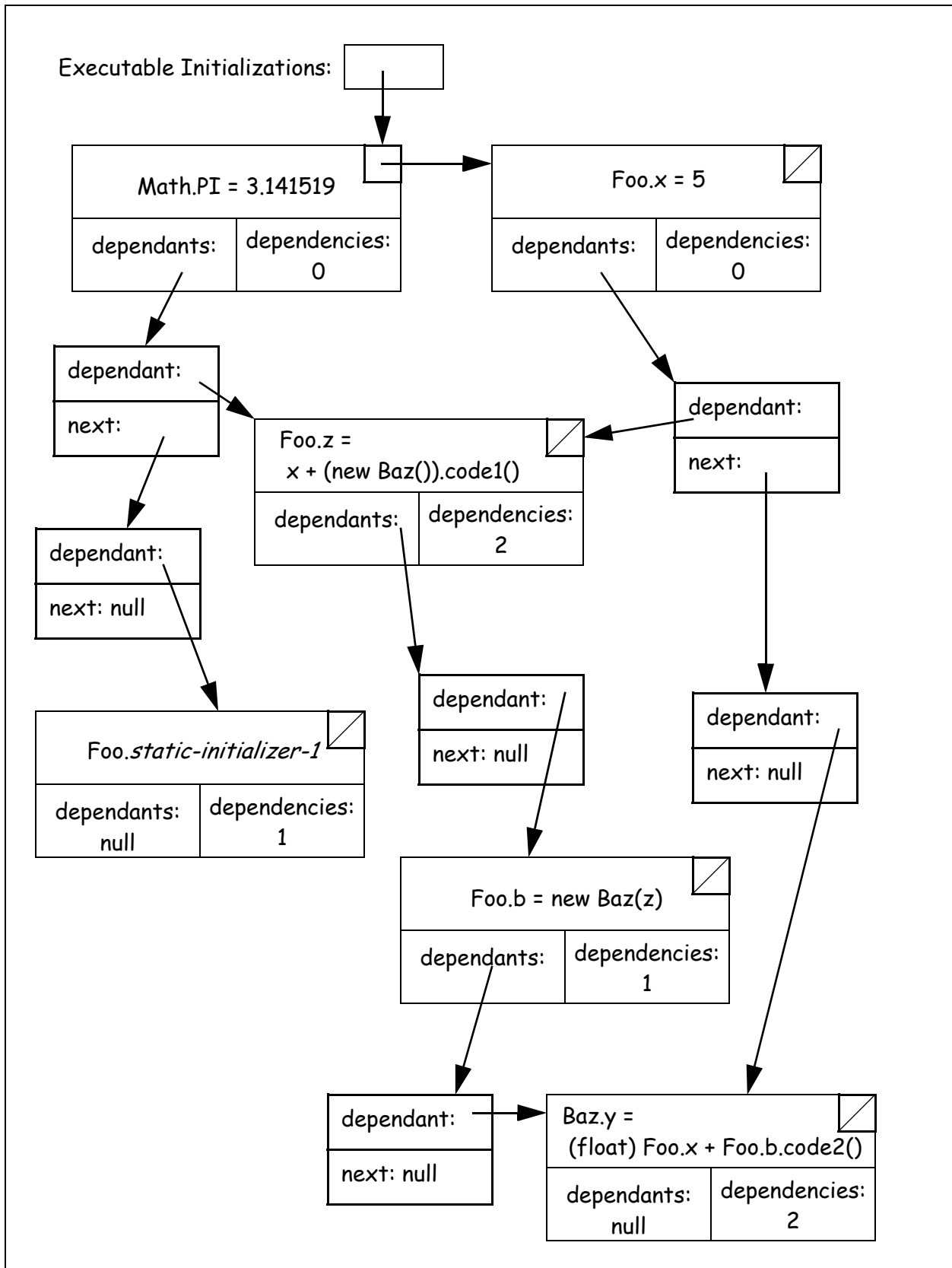


Figure 3: Dependency Graph for Topological Sort

(Permission granted to reproduce and distribute as a complete document, without modification)

In this particular example, the likely initialization sequence would be as follows:

1. Initialize `Math.PI`
2. Initialize `Foo.x`
3. Initialize `Foo.z`
4. Initialize `Foo.b`
5. Initialize `Baz.y`
6. Commit the initial values of the above static class variables to the static load image. Arrange for the startup code to:
  - a. Execute `Foo.static-initializer-1` before invoking the `main` method.

**Discussion.** Note that the `@StaticDependency` and `@InitializeAtStartup` annotations are associated with particular variables rather than the initialization expressions themselves. Furthermore, recognize that it is certainly possible to hide all of the initialization of static class variables in native code that is invisible to the smart linker, and thus cannot be analyzed. It is the developer's responsibility to coordinate with the smart linker by arranging his or her code in ways that can be analyzed. There are various ways to do this.

Suppose, for example, that you've got a single native method that is going to directly initialize an assortment of static class variables. Call these:

```
public static int native_initialized_1, native_initialized_2, native_initialized_3;
```

One way to force the initialization code to execute is by introducing a new placeholder variable, as in:

```
@InitializeAtStartup
private int native_placeholder = the_native_method_that_initializes_everybody();
```

Now, introduce annotations to force this native method to be executed by rewriting the declarations of the first three variables:

```
@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_1;
```

```
@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_2;
```

```
@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_3;
```

At the Belgium meeting of the Open Group, there was some disagreement as to whether the initial main thread should run at the maximum or minimum priority. The choice is a bit arbitrary, as either default behavior can be easily overridden by calling `RealtimeThread.currentRealtimeThread.setPriority()` as the first request within this main method. The benefit of the approach currently drafted is that this gives the safety-critical Java code greater assurance that its main code will not be starved by other higher priority activities running outside the safety-critical Java environment.

**Byte Code Verification.** The following additional byte-code verification is required to support the capabilities described in this section:

**(Permission granted to reproduce and distribute as a complete document, without modification)**

1. For each use of the `@StaticDependency` annotation, the identified class is available as part of the current class path and the named field is a static class variable associated with the identified class.
2. For each use of the `@StaticDependency` and `@InitializeAtStartup` annotations, the associated field declaration describes a static class variable.
3. Each static (class) variable is initialized exactly once, either with an assignment that is part of the variable's declaration or with an assignment statement within the body of a static initializer block.
4. In either case, all initialization code is restricted according to the following rules:
  - a. The code is not allowed to perform any side effects (no synchronization, no starting of threads, no I/O, no increment or decrement operations, and no assignments) except for the following:
    - i. A single assignment appearing outside of any looping control structures must be provided for each class variable.
    - ii. Assignments are allowed to the instance fields of objects that were located in the current evaluation context and which are not reachable from any class variable (i.e. temporary objects).
    - iii. Assignments are allowed to the local variables of a static initialization context.
  - b. Only final methods may be invoked from within initialization code.

## 2. Starting up a Safety-Critical Java Application

When a safety-critical Java program begins execution, it first executes whatever sequence of initialization instructions was deferred until startup time. It does this in a single thread which runs at whatever priority was assigned by the host “operating system” when the safety-critical Java environment was “invoked”. This behavior will differ from one operating environment to the next. We consider this aspect of the system startup to be Implementation Defined.

After initialization of static variables completes, the safety-critical Java environment startups up a main `NoHeapRealtimeThread` running at the highest non-interrupt priority level and arranges for this thread to execute the main method of the “initial class”, which should be declared with the following signature:

```
public static void main(String args[]);
```

At the time this method begins to execute, this is the only thread running in the safety-critical Java environment. The initial class is specified as part of the configuration of the safety-critical Java environment, using Implementation Defined mechanisms.

## 3. Sharing Objects with Traditional Java Domain

The flight mission planning example above helps to motivate the need for and design of the traditional Java API. This section describes the API in more thorough detail. When a hard real-time object is published to the traditional Java world, the system creates a traditional Java object to serve as a proxy for the hard real-time object within the traditional Java domain. When a traditional Java thread invokes a method of a proxy object, the resulting interaction with the hard real-time environment may require the use of a proxy thread that is scheduled according to the rules of the hard real-time environment.

**Declaration of Traditional Java Methods.** A method that is declared with the `@TraditionalJavaMethod` annotation is designed to be invoked by a traditional Java thread running in an environment that is cooperating with the hard real-time environment. Consider, for example, the source code for the hard real-time

(Permission granted to reproduce and distribute as a complete document, without modification)

```
[1] package samples;
[2]
[3] import javax.realtime.util.mc.TraditionalJavaMethod;
[4]
[5] public class Thermostat {
[6]     private float measured_temperature;
[7]     private float desired_temperature;
[8]
[9]     public final @TraditionalJavaMethod @ScopedPure synchronized
[10]         float getTemperature() {
[11]         return measured_temperature;
[12]     }
[13]
[14]     public final @TraditionalJavaMethod @ScopedPure synchronized
[15]         void setTemperature(float f) {
[16]         desired_temperature = f;
[17]     }
[18]
[19]     public final synchronized void updateTemperature(float f) {
[20]         measured_temperature = f;
[21]     }
[22]
[23]     public final synchronized float checkThermostat() {
[24]         return desired_temperature;
[25]     }
[26] }
```

Figure 4: Annotated Source code for **Thermostat** Class

class named `Thermostat`, which is shown in Figure 4. In this class, the `getTemperature()` and `setTemperature()` methods are both identified as traditional Java methods, meaning these methods are to be invoked only by traditional Java threads. When compiled and linked into a hard real-time Java execution environment, the `updateTemperature()` and `checkThermostat()` methods are regular methods for invocation by other hard real-time threads.

**The Registry.** The class `javax.realtime.util.mc.Registry` provides mechanisms for use by hard real-time components that desire to expose certain services to the traditional Java domain. The `Registry` only allows objects that were allocated within scopes that are associated with methods declared to have the `@TraditionalJavaShared` attribute. In theory, a hard real-time Java environment can concurrently share objects with multiple independent Java virtual machine environments. The `@TraditionalJavaShared` annotation takes an optional argument, named `jvm_id`, which specifies the name of the traditional Java virtual machine with which a given scope is allowed to share its allocated objects.

Having instantiated a hard real-time object to be shared with the traditional Java environment, the hard real-time programmer publishes this object so that it can be seen by traditional Java threads by invoking the `publish()` method of `javax.realtime.util.mc.Registry`, passing as arguments the unique string name by which this object will be identified within the registry and a reference to the object. Traditional Java components will obtain references to this object's proxy by invoking the `javax.rtpoxy.Root.lookup()` method within the traditional Java environment, passing the string name that identifies the object as the sole argument to the `lookup()` method.

The following services are supported by the `javax.realtime.util.mc.Registry` abstraction:

`public static Registry instance() throws IllegalStateException`

Returns a reference to the primordial instance of `Registry`, or throws `IllegalStateException` if the primordial instance has not yet been instantiated.

`public static Registry instance(String name) throws IllegalStateException`

Returns a reference to the instance of `Registry` that corresponds to the connected Java virtual machine that is identified by `name`, or throws `IllegalStateException` if no `Registry` has been instantiated to represent an interface to the named virtual machine.

`public Registry(int num_proxy_stacks, int stack_size) throws IllegalStateException`

Constructs the primordial instance of `Registry` having an initial pool of `num_proxy_stacks` `ThreadStack` objects, each having the specified `stack_size`. This method throws `IllegalStateException` if the primordial instance has already been instantiated. This method throws `OutOfMemoryError` error if there is not sufficient memory to instantiate the `ThreadStack` objects.

`public Registry(String name, int num_proxy_stacks, int stack_size)  
throws IllegalStateException`

Constructs an instance of `Registry` to represent the interface to the Java virtual machine identified by `name`, allocating an initial pool of `num_proxy_stacks` `ThreadStack` objects, each having the specified `stack_size`. This method throws `IllegalStateException` if a `Registry` instance has already been instantiated with this same `name`. This method throws `OutOfMemoryError` error if there is not sufficient memory to instantiate the `ThreadStack` objects.

`public final void publish(@Scoped String name, @Scoped Object obj)  
throws IllegalArgumentException, IllegalStateException`

Publish `obj` so it is accessible from the traditional Java domain. Both `name` and `obj` may be scope-allocated. A run-time check assures that `obj` is allocated within a scope that is declared with the `@TraditionalJavaShared` annotation and this `@TraditionalJavaShared` scope is consistent with this `Registry` object, and that the scope of `name` is compatible with assignment to `obj` (i.e. that `name` resides in the same or more outer nested scope as `obj`). This method throws `IllegalArgumentException` if the scope containing `obj` was not annotated with the `@TraditionalJavaShared` annotation or if the scope containing `name` is not properly nested. This method throws `IllegalStateException` if there already exists a published symbol with the same name. The byte-code verifier assures that every scope that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `awaitClearRegistry()`, as described below.

`public final void unpublish(@Scoped Object obj) throws IllegalStateException`

Remove the hard real-time object `obj` from the public registry so that the traditional Java environment can no longer look it up. Note that the traditional Java environment may still have a way to see this object even if it's no longer in the public registry. For example, a traditional Java thread may have looked up the object before it was unpublished and the corresponding proxy object might retain a reference to the object even after the object was removed from the registry. This method throws `IllegalStateException` if the object is not currently found in the registry.

`public final void unpublish(@Scoped String name) throws IllegalStateException`

Remove the hard real-time object known symbolically by `name` from the public registry so that the traditional Java environment can no longer look it up. Note that the traditional Java environment may still have a way to see this object even if it's no longer in the public registry. For example, a

traditional Java thread may have looked up the object before it was unpublished and the corresponding proxy object might retain a reference to the object even after the object was removed from the registry. This method throws `IllegalStateException` if the object is not currently found in the registry.

```
public boolean addProxyThreadStacks(int num_stacks)
```

Add `num_stacks` `ThreadStack` objects to the pool of proxy threads associated with this `Registry` object. The size of each thread's `ThreadStack` is the same size as was specified in the constructor for this `Registry` object. Returns `true` if the stacks were successfully added, `false` if the stacks were not added because, for example, there was not sufficient memory.

```
public final void awaitClearRegistry()
```

This routine is normally called from a method that has declared itself to have the `@TraditionalJavaShared` attribute. If the caller method does not have this attribute, the invocation returns immediately as there can be no objects from this scope visible to the traditional Java domain. If the caller does have the `@TraditionalJavaShared` attribute, the method returns only after all of the proxy objects that were shared from this method's allocation scope have all been removed from the traditional Java domain. In general, this requires that the traditional Java garbage collector has reclaimed all of the proxy objects and the finalizer code for those proxy objects has unregistered the objects from the shared registry. Note that removal of objects from a shared Java registry should be a very rare event. Thus, we are willing to invest some "time and energy" in making sure there is a clean separation of concerns before destroying the corresponding hard real-time allocation context.

**Traditional Java Proxies for Hard Real-Time Objects.** The proxy for a hard real-time instance of `java.lang.Object` is represented in the traditional Java domain by an instance of `javax.rtproxy.Root`. Arrays from the hard real-time environment are represented within the traditional Java domain by proxy objects of one of the following classes: `BooleanArray`, `ByteArray`, `ShortArray`, `CharArray`, `IntArray`, `LongArray`, `FloatArray`, `DoubleArray`, `RefArray`. All of these classes are defined within the `javax.rtproxy` package and each extends `javax.rtproxy.Root`. Each array class supports `length()`, `atGet()`, and `atPut()` methods. For `RefArray`, `length()` always returns zero and the `atGet()` and `atPut()` methods always throw `ArrayIndexOutOfBoundsException` because we do not allow the traditional Java environment to directly fetch or store reference values within the hard real-time environment.

All other classes defined within the hard real-time domain and shared with the traditional Java environment extend `javax.rtproxy.Root` (either directly or indirectly), using a copy of the hierarchy that derives from `java.lang.Object` within the hard real-time domain. Classes defined in the default package of the hard real-time environment are treated as members of the `javax.rtproxy.packages` package within the traditional Java environment. Hard real-time classes defined in named packages are treated as subpackages of `javax.rtproxy.packages` within the traditional Java environment.

**Preprocessing of Traditional Java Methods.** To use the `Thermostat` class shown in Figure 4 as an interface between traditional Java and hard real-time Java components, apply the `rtproxyc` (real-time proxy compiler) program to this file, producing as output one class file that runs in the traditional Java domain and another that runs in the hard real-time domain. The deployment process is illustrated in Figure 5. The source code of Figure 4 is translated into class files representing the program components shown in Figures 6 and 7 respectively.

When the `Thermostat` class is loaded by the hard real-time class loader, the two methods that are tagged with the `@TraditionalJavaMethod` annotation are treated specially. These two methods are invisible to the

(Permission granted to reproduce and distribute as a complete document, without modification)

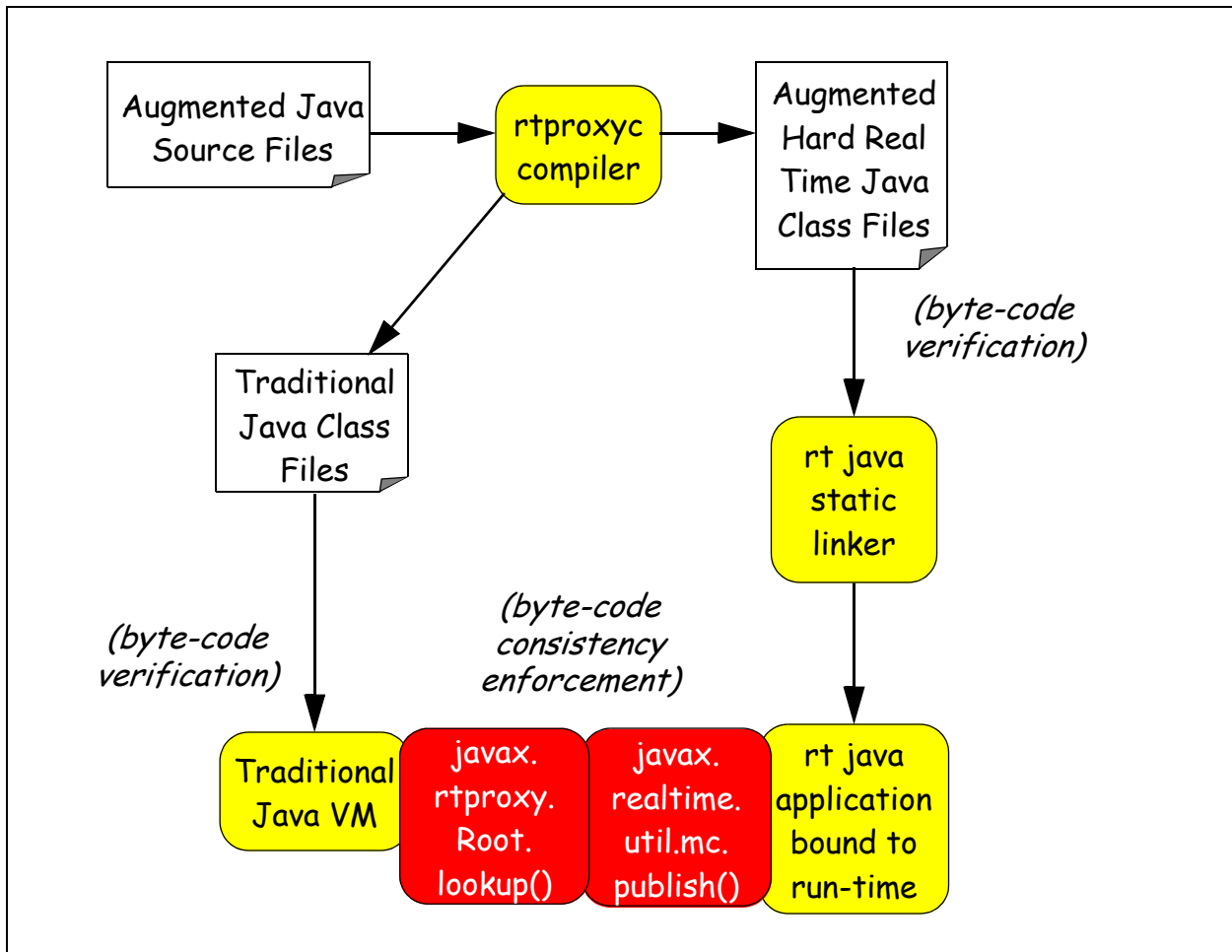


Figure 5: Compilation and Verification of Interface Components

hard real-time domain. Insofar as the hard real-time class hierarchy is concerned, they might as well not be present at all. Instead, these methods are inserted into the body of the proxy class definition that represents instances of this class within the traditional Java environment.

```

[1] package samples;
[2]
[3] public class Thermostat {
[4]     private float measured_temperature;
[5]     private float desired_temperature;
[6]
[7]     public final synchronized void updateTemperature(float f) {
[8]         measured_temperature = f;
[9]     }
[10]
[11]     public final synchronized float checkThermostat() {
[12]         return desired_temperature;
[13]     }
[14] }
  
```

Figure 6: Hard Real-Time View of Thermostat Class

```
[1] package javax.rtpoxy.packages.samples;
[2]
[3] public class Thermostat extends javax.rtpoxy.Root {
[4]     private long handle_to_rt_object;
[5]
[6]     /* Note that these methods are not synchronized even though they were marked
[7]      * synchronized in original source code. That's because the synchronization
[8]      * must be performed according to the rules of the hard real-time environment,
[9]      * so it must be implemented within the native method.
[10]    */
[11]    public final native float getTemperature() {
[12]        try {
[13]            javax.rtpoxy.Libraries.synchronize(handle_to_rt_object);
[14]            // fetch float value at offset 0 within hard real-time object
[15]            f = javax.rtpoxy.Libraries.getFloat(handle_to_rt_object, 0);
[16]        } finally {
[17]            javax.rtpoxy.Libraries.unsynchronize(handle_to_rt_object);
[18]        }
[19]        return f;
[20]    }
[21]
[22]    public final void setTemperature(float f) {
[23]        try {
[24]            javax.rtpoxy.Libraries.synchronize(handle_to_rt_object);
[25]            // store float value at offset 4 within hard real-time object
[26]            javax.rtpoxy.Libraries.setFloat(handle_to_rt_object, 4, f);
[27]        } finally {
[28]            javax.rtpoxy.Libraries.unsynchronize(handle_to_rt_object);
[29]        }
[30]    }
[31] }
```

Figure 7: Traditional Java View of **Thermostat Class**

Since the Java specification requires that stores and fetches to float values be atomic, programmers might be tempted to remove the `synchronized` qualifier from the methods of this class. This would be an error. It turns out that in the absence of the `synchronized` qualifier, the optimization rules for Java compilers would permit the contents of the shared `measured_temperature` and `desired_temperature` variables to be cached in machine registers within the independent contexts of the respective hard real-time and traditional Java threads. Note also that all of the synchronization is performed within the context of the hard real-time execution environment. If a traditional Java thread synchronizes on a proxy object, that will acquire a lock associated with the proxy object within the traditional Java environment. It will have no effect on any locks within the hard real-time domain. The only way to acquire a hard real-time lock is to invoke a traditional Java method and allow that method to acquire the lock using whatever protocols are appropriate for a given implementation of the hard real-time run-time environment.

The implementations of the two traditional Java methods in Figure 7 have been transformed from the original Java source code in order to provide access to the hard real-time object from within the traditional Java environment. The implementation of the `javax.rtpoxy.Libraries` classes may use JNI code, or might use implementation-specific optimizations such as special proxy-aware JIT compilation and in-lining techniques. The methods of the `javax.rtpoxy.Libraries` class have the ability to reach into the hard real-time

domain in order to read and modify objects residing in that domain. These are privileged operations that should not be exposed to arbitrary Java components. In situations where a trusted JIT compiler integrates the implementation of these methods, the JIT compiler assures that these libraries are only invoked from within classes that derive from `javax.rtpoxy.Root`. When these library services are invoked “out of line”, a security check on the stack backtrace is performed to assure that the calling method resides within a class that derives from `javax.rtpoxy.Root`. Any other invocations of these library services represent security violations and are prohibited. A special exception, not specified in the current draft, is thrown.

For purposes of enforcing consistency between hard real-time components and the traditional Java components, the hard real-time class file components include as encoded attributes a representation of the traditional Java methods that were extracted from the original source file. In addition to enforcing the verification rules of traditional Java and the special verification rules of the hard real-time Java profile, the byte-code verifier for the hard real-time class file assures that these traditional Java methods comply with all of the special restrictions imposed by this draft specification on the bodies of such methods. Each time a new proxy class is loaded into the traditional Java domain, we perform a consistency check to assure that the methods of the traditional Java proxy object implement exactly the same functionality that is specified in the corresponding hard real-time class file’s attribute encodings.

**Execution of Traditional Java Methods.** Proxy objects within the traditional Java domain are created each time a new hard real-time object is published by way of the registry and each time a reference to a new hard real-time object is returned from a traditional Java method. Every proxy is either an instance of `javax.rtpoxy.Root` or some class that extends this class.

When a reference value is returned or thrown from a traditional Java method, it is replaced within the traditional Java environment by its corresponding proxy. If no proxy exists, one is created at the time the value is returned. Reference values communicated from the hard real-time domain to the traditional Java domain must either reside within `ImmortalMemory` or within scopes that were declared with the `@TraditionalJavaShared` annotation. When a traditional Java method returns or throws a reference, a run-time check is performed at the interface to the traditional Java domain to assure that the reference corresponds to an object that can be shared with the traditional java environment, as indicated by the corresponding `@TraditionalJavaShared` annotation. It is the hard real-time programmer’s responsibility to assure this requirement is satisfied. If a violation is detected, the traditional Java method terminates by throwing an `IllegalStateException`.

When a proxy object is passed as an argument to a traditional Java method, it is replaced with a reference to the proxy object’s hard real-time referent while the method executes in the hard real-time domain. If a traditional Java method takes one or more reference arguments, it is necessary to assure that all of the reference arguments and the hard-real-time equivalent of the proxy’s `this` are all scope compatible. This check consists of identifying which of the reference arguments is most deeply nested and then assuring that all of the other references correspond to scopes that are more outer nested on the same stack as the most deeply nested scope. If this condition is not satisfied, the traditional Java method invocation aborts, throwing an instance of `IllegalArgumentException` to the traditional Java thread that attempted to perform this invocation.

The constructor for `javax.rtpoxy.Root` includes a check to enforce that the only thread allowed to instantiate an instance of `javax.rtpoxy.Root` or of any of its subclasses is the special thread that is responsible for maintaining the relationship between the traditional Java domain and the hard real-time environment. This guarantees that any instances of `javax.rtpoxy.Root` and instances of any of its subclasses are true proxies for hard real-time objects residing in the hard real-time environment.

By applying this technique to assure that every proxy object was instantiated under our direction, and the implementation of its methods was derived from code written by the hard real-time developer, and the code has successfully fulfilled the stringent requirements of the hard real-time Java byte code verifier, there is no need for additional run-time checks on the invocation of the proxy object's traditional Java methods, except when a traditional Java method expects one or more reference arguments.

These conventions guarantee that the hard real-time domain never holds a reference to a traditional Java object, entirely eliminating the risk that hard real-time threads might need to coordinate with execution of the garbage collector. This avoids the problems that preclude synchronization between hard real-time and non-real-time threads in the standard RTSJ.

**Proxy Thread Management.** When a traditional Java thread invokes a traditional Java method, the body of the method is executed at the priority of the Java thread that is executing. This thread is free to detach from the traditional Java virtual machine while it is executing the hard real-time Java method because it is guaranteed that no garbage collected objects will be accessed by the thread while it is executing this code.

If the traditional Java method performs synchronization, the priority inversion avoidance implementation is under the control of the hard real-time scheduler. If the synchronization lock is governed by priority ceiling emulation, the thread will immediately elevate its priority to the ceiling associated with that lock. If the lock is controlled by priority inheritance, this thread becomes eligible to inherit the higher priorities of the hard real-time threads. In both cases, it is important to recognize that even though the traditional Java thread holds a hard real-time lock, it only does so under the conditions that:

1. It has already detached from the JVM's scheduler and is not interacting with the JVM's garbage collector, and
2. It is only executing code that was written by the hard real-time developer.

Whenever a traditional Java method performs synchronization, the traditional Java thread is replaced by a proxy thread running within the hard real-time domain. Memory for the proxy thread is provided from the pool of proxy `ThreadStack` objects maintained by `javax.realtime.util.mc.Registry`. The proxy's stack is instantiated as if a new thread had been spawned from the scope the contains the mostly deeply nested argument passed to the traditional Java method. In the case that the traditional Java method invocation does not pass any reference arguments, the allocation context of `this` is by default the mostly deeply nested context. This assures that the rules of referential integrity will allow the proxy thread's local variables and any objects allocated on the proxy thread's stack to make reference to objects that are contained within outer-nested scopes.

### Byte Code Verification

The following additional byte code verification checks are required to support the capabilities described in this proposal:

1. The byte-code verifier assures that every scope that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `awaitClearRegistry()` as the last statement in the `finally` clause corresponding to the `try` clause that contains all executable code except the clean up code associated with the method. The clean up code, all of which must appear in the `finally` clause, consists only of:
  - a. Invocations of `ThreadStack.join()`,
  - b. Invocations of `Registry.unpublish()`, and
  - c. A single invocation of `Registry.instance().awaitClearRegistry()`. If the enclosing method's `@TraditionalJavaShared` annotation specifies the name of a particular Java virtual machine, the

**(Permission granted to reproduce and distribute as a complete document, without modification)**

byte-code verifier assures that the same name is supplied as an argument to this `instance()` invocation.

2. Any invocation of `ThreadStack.spawn()` is contained within a `try` statement, and the corresponding `finally` statement includes a matching invocation of `ThreadStack.join()` for the same `ThreadStack` object.
3. Any method invoked from within a traditional Java method must be declared with the `@TraditionalJavaMethod` attribute.
4. Every traditional Java method must be declared with the `@ScopedPure` annotation and must not have the `@CallerAllocatedResult` annotation.
5. A traditional Java method only accesses instance fields of its own object (`this`), static variables associated with its own class, and the elements of arrays that are referenced from its local, instance, or class static variables. It is not allowed to access instance fields of other objects or static fields of other classes.

#### 4. Annotation to Support Static Analysis of Hard Real-Time Components

This section describes annotations that allow programmers to clarify their intentions with respect to static properties of hard real-time software components.

##### `@StaticAnalyzable`

A method or constructor identified with this attribute must conform to certain style guidelines that make it possible to analyze worst-case execution time, stack memory allocation, and heap memory allocation, and to prove that this program component will not invoke any operations that might potentially block. If a method is declared to be `StaticAnalyzable`, all overriding methods must also be `StaticAnalyzable` and must enforce analysis for the same execution modes.

A class may be annotated with this attribute if it is also annotated with `@ReentrantScope` or `@NestedReentrantScope` annotations. In this case, the value of the `modes` attribute should be the same as for each and every constructor for the class. The `enforce_time_analysis` and `enforce_non_blocking` attributes have no meaning when associated with a class. The `enforce_memory_analysis` attribute signifies that the total size of the reentrant scope can be determined through static analysis of this object's various memory-allocating virtual methods.

##### `@StaticAnalyzable(enforce_time_analysis = {false})`

This method annotation denotes that the method should be analyzed, and requires that its stack and heap memory allocation be analyzable, and that the body of the method not invoke any blocking operations, but does not require that its CPU time requirements be analyzable. In this case, the analysis infrastructure simply gathers as much CPU time information as is available. Note that CPU time requirements do not include the time spent waiting to acquire locks. That time must be analyzed independently.

```
@StaticAnalyzable(
    enforce_time_analysis = {false, true, true, true},
    modes = ModeEnumeration.class
)
```

Assume the following declaration of `ModeEnumeration`:

```
public enum ModeEnumeration { UNBOUNDED, SMALL, MEDIUM, LARGE }
```

The method annotation denotes that the method has four different modes of operation. The CPU time requirements for the first mode, identified as `UNBOUNDED`, need not be analyzable. The CPU time requirements for the other three modes must be analyzable. In all cases, the stack and heap memory allocation requirements must be analyzable and the method must not block.

```
@StaticAnalyzable(  
    enforce_time_analysis = {false, true, true, true},  
    enforce_non_blocking = {false, true, true, true},  
    enforce_memory_analysis = {false, true, true, true},  
    modes = ModeEnumeration.class  
)
```

This annotation indicates that the method's CPU time and memory allocation requirements must be statically analyzable for the `SMALL`, `MEDIUM`, and `LARGE` modes of execution. Furthermore, it is required that the static analyzer demonstrate that this method will not block in any of these three modes of execution. This means the method may be invoked from within an `Atomic` synchronized context, including the body of an interrupt handler.

We require that all overriding methods are annotated with the same `StaticAnalyzable` annotation, with the same value for the `modes` attribute. For any `false` entry in the `enforce_time_analysis`, `enforce_non_blocking`, or `enforce_memory_analysis` arrays, the overriding method may have a corresponding value of `true`. For any `true` entry in one of these arrays, the overriding method must also have a `true` value.

#### @DelegatedAnalyzable

This annotation accompanies methods that are intended to be statically analyzable, but which cannot be analyzed independent of the context from which they are invoked. For example, a generic method to sort the entries of a collection may be analyzable if we know the size of the collection and we know the cost of invoking the comparator. But the comparator's cost may be different for each different type of collection. In this situation, the `sort()` method might be declared as `@DelegatedAnalyzable`, with an explicit dependency on knowing the behavior of the comparator function. The attributes associated with the `@DelegatedAnalyzable` annotation are the same as for `@StaticAnalyzable`. If a method is declared as `@DelegatedAnalyzable`, the byte-code verifier assures that the method's behavior can be analyzed in full accordance with the values of its `modes`, `enforce_time_analysis`, `enforce_memory_analysis`, and `enforce_non_blocking` attributes, conditioned upon resolution of the delegator's behavior with respect to these same attributes.

#### @Delegator

The `@Delegator` annotation names variables whose virtual methods may determine static analyzability of a `@DelegatedAnalyzable` method. Use the `@Delegator` annotation on selected arguments to the `@DelegatedAnalyzable` method and on up to one `final` instance variable of the object to which the `@DelegatedAnalyzable` method belongs. Whenever a class defines a `final` instance variable that is declared with the `@Delegator` annotation, the byte-code verifier requires that every constructor for the class initialize this variable by copying the value from a `@Delegator` argument to the constructor. This `@Delegator` argument must have the same type as the `@Delegator` instance variable.

#### @Delegate

The `@Delegate` annotation serves to help the static analysis tools recognize the patterns of usage with respect to invocation of `@DelegatedAnalyzable` methods. If analysis of a particular object's `@StaticAnalyzable` methods depends upon the ability to analyze certain `@DelegatedAnalyzable`

methods which themselves make call-back invocations to `@StaticAnalyzable` methods of the first object, the programmer is expected to establish a permanent link between the two objects at the time of the object's construction. The static analyzable pattern consists of the following:

- A. The original (`@Delegator`) object's constructor instantiates the second (`@Delegate`) object, passing a reference to itself as the value of the second object's `@Delegator` argument.
- B. The second (`@Delegate`) object's constructor copies the value of its `@Delegator` argument to the single `final` instance field that is named with the `@Delegator` annotation.
- C. Upon return from the second (`@Delegate`) object's constructor, the first object's constructor copies the returned reference value to a `final` instance fields that is named with the `@Delegate` annotation. The type of the `@Delegate` instance fields must exactly match the type of the constructed object.
- D. Every constructor for the second (`@Delegate`) object must initialize the same `final` instance `@Delegate` fields with instances of the same classes. In other words, `@Delegate` field  $x$  is always assigned a reference to a newly constructed instance of class  $U$ . `@Delegate` field  $y$  is always assigned a reference to a newly constructed instance of class  $V$ . And so on.
- E. Static analysis of the first (`@Delegator`) object's `@StaticAnalyzable` methods may make depend on analysis of `@DelegatedAnalyzable` methods of the the second (`@Delegate`) object which are invoked as virtual methods associated with one of this object's `@Delegate` instance variables.. Analysis of the second (`@Delegate`) object's `@DelegatedAnalyzable` methods may in turn depend on analysis of the first (`@Delegator`) object's `@StaticAnalyzable` methods, which are invoked as virtual methods associated with that object's `@Delegator` instance variable. Direct and indirect recursion of individual `@DelegatedAnalyzable` methods is prohibited by the byte-code verifier. In the context of analyzing the first (`@Delegator`) object's `@StaticAnalyzable` methods, all of the information required to perform the static analysis is fully resolved. It is important to note that the `@DelegatedAnalyzable` methods themselves may be invoked from various different contexts, and the resource needs associated with those methods may differ for each context from which it is invoked.
- F. Note that we do not require that every method that invokes a `@DelegatedAnalyzable` method itself be static analyzable. If a delegating object does not care to enforce static analyzability of its methods, that is fine. It can still invoke the `@DelegatedAnalyzable` methods of the "generic" delegate object. In this circumstance, we do not require that the delegating object's constructors consistently initialize any `final` instance variable annotated with the `@Delegate` attribute.

#### `StaticLimit.InvocationMode(Enum callee_mode)`

This assertion always returns `true`. Place this as an assertion within the body of a `@StaticAnalyzable` method, immediately preceding an invocation of another method to specify that the other method, when called from this particular context, should be analyzed according to its mode `callee_mode`. If no `InvocationMode()` assertions precede invocation of the method, it is assumed that the method runs in its default mode, which is always the first mode in its corresponding mode enumeration.

**StaticLimit.InvocationMode(Enum caller\_mode, Enum callee\_mode)**

This assertion always returns **true**. Place this as an assertion within the body of a **@StaticAnalyzable** method, immediately preceding an invocation of another method to specify that the other method, when called from this particular context, should be analyzed according to its mode **callee\_mode** if this method is being analyzed in mode **caller\_mode**. If no **InvocationMode()** assertions precede invocation of the method, it is assumed that the method runs in its default mode, which is always the first mode in its corresponding mode enumeration. If multiple **InvocationMode()** assertions precede invocation of a method, each one having a different value of the **caller\_mode** attribute, the first one to match the **caller\_mode** is applied.

**StaticLimit.InvocationBound(int bound)**

This assertion represents the maximum number of times the enclosing method will be invoked for each instantiation of the enclosing class. This is most relevant when implementing methods of a class that is declared with the **@ReentrantScope** or **@NestedReentrantScope** and **@StaticAnalyzable** annotations. For all such classes, this assertion should appear within each method that allocates memory within the reentrant scope in a position that dominates all exits from the method. For each such method, the total amount of memory allocated during each method invocation must be statically analyzable from analysis of the class body alone, without necessarily knowing the modes of execution for each method invocation.

**StaticLimit.InvocationBound(Enum constructor\_mode, int bound)**

This assertion represents the maximum number of times the enclosing method will be invoked for each instantiation of the enclosing class, given that the class was instantiated by invoking a constructor with execution mode matching **constructor\_mode**. This is most relevant when implementing methods of a class that is declared with the **@ReentrantScope** or **@NestedReentrantScope** and **@StaticAnalyzable** annotations. For all such classes constructed with execution modes for which **enforce\_memory\_analysis** is **true**, this assertion should appear within each method that allocates memory within the reentrant scope in a position that dominates all exits from the method. For each such method, the total amount of memory allocated during each method invocation must be statically analyzable from analysis of the class body alone, without necessarily knowing the modes of execution for each method invocation.

**StaticLimit.IterationBound(int max\_iterations)**

This assertion enforces that the inner-most enclosing loop iterates through this particular statement no more than **max\_iterations** times for each time the loop itself is entered. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered.

**StaticLimit.IterationBound(Enum analysis\_mode, int max\_iterations)**

This assertion enforces that the inner-most enclosing loop iterates through this particular statement no more than **max\_iterations** times for each time the loop itself is entered when the enclosing method is analyzed according to **analysis\_mode**. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered.

**StaticLimit.NestedIterationBound(int nesting\_level, int max\_iterations)**

This assertion enforces that the outer nested enclosing loop at nesting level **nesting\_level** iterates through this particular statement no more than **max\_iterations** times for each time the loop itself is entered. If this assertion appears nested within conditionally executed code, it serves to limit the

number of times the conditionally executed context will execute for each time the enclosing loop is entered. `StaticLimit.NestedIterationBound(0, max_iterations)` is identical to `StaticLimit.IterationBound(max_iterations)`.

`StaticLimit.NestedIterationBound(Enum analysis_mode, int nesting_level, int max_iterations)`  
This assertion enforces that the outer nested enclosing loop at nesting level `nesting_level` iterates through this particular statement no more than `max_iterations` times for each time the loop itself is entered when the enclosing method is analyzed according to `analysis_mode`. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered. `StaticLimit.NestedIterationBound(analysis_mode, 0, max_iterations)` is identical to `StaticLimit.IterationBound(analysis_mode, max_iterations)`.

`StaticLimit.NotReached()`

This assertion always returns `false`. It denotes to the execution-time analyzer that this particular statement should never be executed. Place this assertion immediately after a method invocation that is expected to always throw an exception, for example.

`StaticLimit.NotReached(Enum analysis_mode)`

This assertion always returns `false`. It denotes to the execution-time analyzer that this particular statement should never be executed when the enclosing method is analyzed according to `analysis_mode`. Place this assertion immediately after a method invocation that is expected to always throw an exception, for example.

`StaticLimit.ArrayLength(byte ba[], int bound)`

This assertion appears immediately following an allocation of a byte array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, byte ba[], int bound)`

This assertion appears immediately following an allocation of a byte array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(char ca[], int bound)`

This assertion appears immediately following an allocation of a character array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, char ca[], int bound)`

This assertion appears immediately following an allocation of a character array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(short sa[], int bound)`

This assertion appears immediately following an allocation of a short (16-bit) integer array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, short sa[], int bound)`

This assertion appears immediately following an allocation of a short (16-bit) integer array. It

establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(int ia[], int bound)`

This assertion appears immediately following an allocation of a 32-bit integer array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, int ia[], int bound)`

This assertion appears immediately following an allocation of a 32-bit integer array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(float fa[], int bound)`

This assertion appears immediately following an allocation of a 32-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, float fa[], int bound)`

This assertion appears immediately following an allocation of a 32-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(long la[], int bound)`

This assertion appears immediately following an allocation of a 64-bit long integer array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, long la[], int bound)`

This assertion appears immediately following an allocation of a 64-bit long integer array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(double da[], int bound)`

This assertion appears immediately following an allocation of a 64-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, double da[], int bound)`

This assertion appears immediately following an allocation of a 64-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

`StaticLimit.ArrayLength(object oa[], int bound)`

This assertion appears immediately following an allocation of an array of references. It establishes an upper bound on the number of entries in the newly allocated array.

`StaticLimit.ArrayLength(Enum analysis_mode, object oa[], int bound)`

This assertion appears immediately following an allocation of an array of references. It establishes

an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to `analysis_mode`. If multiple `ArrayLength()` assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

The `@StaticAnalyzable` annotation has several additional attributes, not described above. These attributes are to be calculated automatically by the development environment during class loading. The byte-code verifier prevents the programmer from overriding these values. The fields may be consulted by static analysis tools for purposes of constructing static schedules, and during static initialization for purposes of enforcing resource limits. The fields are:

`long [] execution_time()`

Represents the nanoseconds required to execute this method's code in each of the identified modes of operation. There is one entry in this array for each of the enumeration constants in the `modes()` Enum type.

`long [] stack_bytes()`

Represents the maximum number of bytes of stack growth required during execution of this method for each of the identified modes of operation. There is one entry in this array for each of the enumeration constants in the `modes()` Enum type.

`long [] heap_bytes()`

Represents the maximum number of bytes of heap growth required during execution of this method for each of the identified modes of operation. We use the term "heap" loosely. This does not refer to Java's garbage-collected heap. Rather, it refers to the explicitly managed allocation contexts that are used for hard real-time memory allocation. There is one entry in this array for each of the enumeration constants in the `modes()` Enum type.

There are several constants defined in the `StaticLimit` class. These are listed below:

`public enum DefaultAnalysisMode { CONSERVATIVE }`

This is the default enumeration supplied as the value of the `modes` attribute of an `@StaticAnalyzable` annotation.

`public static final long UNANALYZABLE_TIME`

When used to represent the value of a `@StaticAnalyzable execution_time` attribute, `UNANALYZABLE_TIME` means the corresponding program component does not follow the guidelines that are required to support automatic analysis of worst-case execution time.

`public static final long UNANALYZED_TIME`

When used to represent the value of an `@StaticAnalyzable execution_time` attribute, `UNANALYZED_TIME` means the corresponding program component does follow the guidelines that are required to support automatic analysis of worst-case execution time, and thus the worst-case execution time is analyzable. However, due to limits of the development environment and/or run-time environment, this program component has not been analyzed.

`public static final long UNANALYZABLE_SIZE_BYTES`

When used to represent the value of a `@StaticAnalyzable stack_bytes` or `heap_bytes` attribute, `UNANALYZABLE_SIZE_BYTES` means the corresponding program component does not follow

the guidelines that are required to support automatic analysis of worst-case memory allocation needs.

```
public static final long UNANALYZED_SIZE_BYTES
```

When used to represent the value of an `@StaticAnalyzable` `stack_bytes` or `heap_bytes` attribute, `UNANALYZED_SIZE_BYTES` means the corresponding program component does follow the guidelines that are required to support automatic analysis of worst-case memory allocation needs. However, due to limits of the development environment and/or run-time environment, this program component has not been analyzed.

## 5. Annotations to Support Maintenance of Multi-Linked Data Structures

Because of the restriction that no pointer may refer from an object residing at a shallow stack location to an object residing in a deep stack location, it is especially difficult to work with dynamically evolving data structures that include bi-directional links between the objects that comprise the data structure. If newly allocated objects contain links only to previously allocated objects, then the scoped-memory assignment rules are easily satisfied. However, when previously allocated objects may need to maintain references to newly allocated objects, the traditional scoped-memory assignment rules will be violated.

Some examples of data structures that are likely to violate the referential integrity scoping rules include:

- A deque (doubly-ended queue) in which each entry on the queue maintains both a next and previous pointer.
- An AVL tree, in which new entries are always added as leaves, but the rebalancing process might result in a newer entry bubbling up to assume a position closer to the root of the tree.
- An open hash table, in which hash collisions are resolved by building a linked list to represent the conflicting nodes.
- A `StringBuffer` object that must allocate additional buffer space in order to extend the length of its `String` data.
- Just about any of the classes in the `java.util.Collection` hierarchy.

These are just a few examples of abstractions that will behave badly if the data structure is initially constructed in one context and new elements allocated from inner-nested contexts are subsequently added to the original data structure.

One practice that has been recommended to developers who must manage multi-linked data structures within the context of the Real-Time Specification for Java's scoped memory abstractions is to perform the following each time a new element must be added into the data structure:

1. Use `MemoryArea.getMemoryArea()` to find the scope that contains the the parent data structure, by passing as an argument a reference to the parent data structure.
2. Use `ScopedMemory.newInstance()` to instantiate the desired new element within the same scope as the parent data structure.
3. Insert the new element into the shared data structure.

This will work in practice. However, this violates software engineering principles of data abstraction and encapsulation. In particular, all users of the data structure are held responsible for following this memory management protocol. If they feel to do so, the data structure will become compromised. From a software

engineering perspective, it is much better for the implementor of the data abstraction to take responsibility for assuring that the referential integrity rules are followed in order to preserve the integrity of the data structure.

Besides ensuring that the methods that operate on the shared data structure do not result in throwing of an `IllegalAssignmentException`, another responsibility of the data structure's implementor is to arrange that the memory scope within which the data structure is allocated is sufficiently large to represent all of the objects that comprise the data structure. This responsibility is very foreign to typical Java programmers, but it is an essential part of a hard real-time developer's job description. Ultimately, in order to bound the scope size, developers must impose limits on the amount of information that is to be represented in each such data structure.

Documenting and enforcing these data structure limits are critical aspects of developing and maintaining composable real-time software components. The practice of allowing client software at great logical distance from the implementation of a particular data structure's abstract services to reach back into the data structure's scope and allocate new objects within that scope contradicts both of these objectives. Using static analysis tools to automatically enforce size limits on data structures that can be so easily manipulated from afar is not possible because the problems that must be solved are theoretically "undecidable".

Instead, a more structured syntax is required. The `@ReentrantScope` annotation is designed to specifically support this need. This annotation, which is attached to a Java class definition, introduces the following semantics:

1. When instantiated, this object will either be instantiated on the run-time stack, or in `ImmortalMemory`. If the instantiation resides in `ImmortalMemory`, then all objects allocated by the instance's methods will also reside in `ImmortalMemory`. If, however, the new instance resides within a dynamic scope, it will be important to assure that references to this newly allocated object do not leak into `ImmortalMemory` locations. Note that in order for this object to have been scope-allocated, its constructor must have been declared with the `@ScopedThis` annotation, and the variable to which the constructor's result is assigned must be considered by the compiler to have the `@Scoped` attribute. Thus, we are assured that references to this `@ReentrantScope` object will not survive beyond the lifetime of the object itself.
2. When the class is instantiated, a `MemoryArea` is identified to hold all of the objects to be allocated by the instance methods of this `@ReentrantScope` object. If this object was allocated in `ImmortalMemory`, then the identified `MemoryArea` is `ImmortalMemory`. Otherwise, create a new `ScopedMemory` context at the current scoping level to serve as the common `MemoryArea` within which all objects allocated by this instance's methods are to be allocated. It is important that this new `ScopedMemory` context be at the same nesting level as the `MemoryArea` that contains `this`, because `this` needs to be able to refer directly to the objects allocated by this object's instance methods<sup>1</sup>. There are three alternative ways to determine the size of the new `ScopedMemory` context:
  - a. If the class itself was declared with the `@StaticAnalyzable` annotation, then the static analyzer can automatically determine the total required scope size.

---

1. When translating the `@ReentrantScope` semantics for execution on a vanilla RTSJ platform, the reentrant scope is the same scope as the context that constructs the `@ReentrantScope` object. That shared scope must be large enough to represent all of the objects allocated within either environment. If an enhanced full-RTSJ platform provides a way to expand scopes and/or to partition memory within a scope, the translation of the `@ReentrantScope` constructs would ideally exploit these enhanced capabilities.

- b. Or, the programmer may provide a `@ScopedMemorySize` annotation associated with the class, which denotes the total required scope size.
- c. Or, each constructor may include a single invocation of `javax.realttime.util.sc.SizeEstimator.ensureScopeCapacity()` to specify the desired size of the shared `ScopedMemory` region.

The byte-code verifier assures that for every class annotated with the `@ReentrantScope` attribute, exactly one of these three methods for determining the scope size is valid. In particular, the verifier assures that:

- a. The class is declared as `@StaticAnalyzable`, and each instance method is declared as `@StaticAnalyzable` with the `enforce_memory_analysis` attribute `true` for all analysis modes, and each such method includes an invocation of `StaticLimit.InvocationBound()` dominating all exits from the method, or
- b. The class is not declared as `@StaticAnalyzable` but the class is declared with a `@ScopedMemorySize` annotation, or
- c. The class is not declared with either `@StaticAnalyzable` or `@ScopedMemorySize` annotations. In this case, the verifier assures that every constructor contains exactly one invocation of `javax.realttime.util.sc.SizeEstimator.ensureScopeCapacity()` which dominates every exit from the constructor.

Furthermore, the byte-code verifier assures that for any class that is declared with the `@ReentrantScope` annotation, all subclasses also carry this annotation. However, each subclass may use a different mechanism to determine the size of the shared reentrant scope.

3. Assuming this reentrant-scope object is allocated within scoped memory, the scope number for this newly allocated scope is the same as the scope number for the method that instantiates this object. It is as if to say we are expanding that method's scope. Both scopes will be destroyed at the same time, when the allocating method returns.
4. Whenever an instance method of this object is invoked, the method executes within the original allocation context. Any new objects allocated within this method will reside within the shared `@ReentrantScope` region. None of the memory for these objects will be reclaimed until control returns from the outer-most context (the one that originally instantiated this object).
5. If one of this object's instance methods invokes another method, that new method's scope is numbered one more than the most deeply nested context on the given thread's scope stack. Suppose, for example, that a `@ReentrantScope` is instantiated at scope level 10, and subsequently, the stack has expanded to scope level 18, at which point one of the `@ReentrantScope` object's instance methods is invoked. During execution of the invoked instance method, we are running in scope 10. If this method calls another method, that new method will create a new activation frame and number it as nesting level 19.
6. Within a `@ReentrantScope` object's instance methods, the byte-code verifier treats newly allocated objects in the same way that it treats `@CallerAllocatedResult` objects. In particular, it recognizes that the newly allocated objects are likely to reside in contexts that are much more shallow than certain other scoped contexts that currently exist. This means that any incoming `@Scoped` arguments are likely to point to objects residing in more deeply nested scopes than any objects newly allocated within the instance method. Thus, assignments of `@Scoped` parameter values to these newly allocated objects will need to be checked. If such assignments exist, the byte-code verifier will reject the method outright unless it is declared with the `@AllowCheckedScopedLinks` attribute. In the absence of assignments directly from `@Scoped` parameter values to newly allocated objects, the `@AllowCheckedScopedLinks` annotation is not required. In particular, it is possible to pass a `@Scoped`

argument to a called method, and have that called method instantiate a stand-alone `@CallerAllocatedResult` object (one that does not refer to any of the more inner-nested objects) which can reliably be placed into a collection data structure that resides within the `@ReentrantScope`.

7. Static analysis of a `@ReentrantScope` class's memory requirements depends on cumulative analysis of all methods in the class. In the case that we are analyzing the memory required for the shared scope associated with a `@ReentrantScope` class, besides requiring annotations for "standard" analysis, it is also necessary to annotate each method with a `StaticLimit.InvocationBound()` assertion. This represents the maximum number of times this method will be invoked for any given instantiation of this `@ReentrantScope` object. A single `StaticLimit.InvocationBound()` assertion should appear within the body of each instance method, in a location that dominates all exits from the method. If there are multiple analysis modes specified for the class constructor, the constructor's analysis mode governs interpretation of the `StaticLimit.InvocationBound()` assertions.
8. We require that the byte-code verifier for compliant implementations of the safety-critical Java specification implement the following algorithm in analyzing `@ReentrantScope` classes:
  - a. Initially, consider all `private @Scoped` instance variables to be members of the *safe-set*.
  - b. Repeat until the inner loop completes without making any changes to the *safe-set* membership:
    - i. For each assignment to a variable in the *safe-set*, consider all of the reaching definitions
    - ii. If the origin of the value assigned to this variable is not a new-memory allocation within an instance method, and is not a copy from another private variable within the *safe-set*, remove this variable from the *safe-set*.

For any assignment to variables within the *safe-set*, no scoped-memory assignment checking should be performed. Furthermore, the byte-code verifier should not require the `@AllowCheckedScopedLinks` annotation to accompany such assignments.

In some cases, a `@ReentrantScope` object finds it necessary to make reference to an inner-nested `@ReentrantScope` data structure, expecting the inner-nested object's data structures to reside at the same scope level as the outer-nested `@ReentrantScope` object. As a motivating example, consider a situation in which a `@ReentrantScope ClassLoader` object needs to use multiple `HashMap` data structures to keep track of loaded classes and the desired annotation status of certain named packages and unloaded classes. The hash tables presumably need to reference certain objects that are allocated within the `ClassLoader`'s scope, such as certain string names and even the loaded `Class` data structures.

To address this particular need, we introduce a `@NestedReentrantScope` annotation. A `@NestedReentrantScope` object can only be instantiated from within a constructor of a `@ReentrantScope` or `@NestedReentrantScope` object. The `@NestedReentrantScope` object is like a `@ReentrantScope` in that it satisfies all of the constraints described above. The main differences implemented in `@NestedReentrantScope` objects are the following:

1. Incoming `@Scoped` and `@ScopedArray` arguments (but not necessarily `@ScopedLocal` or `@ScopedArrayLocal` arguments) to the instance methods of a `@NestedReentrantScope` object are known to reside in the same scope level as the object's reentrant scope. This is enforced at the point of each invocation by the byte-code verifier. Given this, the instance method's implementation can create references directly to incoming arguments, without having to make a copy, and without requiring a run-time check.
2. When an instance method of a `@NestedReentrantScope` object returns a `@Scoped` or `@ScopedArray` result, the caller is guaranteed that this returned result resides in a scope at the same level as the

invoked object's reentrant scope. This is enforced by the byte code verifier through analysis of reaching definitions for each instance method's return values. Given this, a `@ReentrantScope` object that invokes an instance method of an inner-allocated `@NestedReentrantScope` object can reliably create references between objects allocated within its own scope and the value returned from the instance method of the inner-allocated `@NestedReentrantScope` object.

To support reliable integration of `@ReentrantScope` and `@NestedReentrantScope` objects, a variety of special byte-code verification rules are enforced, as described in section 6 of the main document. This section begins on page 17.

## 6. Annotations to Support Static Enforcement of Referential Integrity

The following JDK 1.5 meta-data annotations enable reliable stack allocation of Java objects. Assume availability of a special byte-code verifier to enforce consistency between annotations and implementation.

### `@ScopedThis`

(applies to methods and constructors)

This annotation indicates that the method's treatment of its implicit `this` argument is consistent with the rules required to allow `this` to refer to an object that resides on the run-time stack of the currently executing thread. In other words, `this` is never assigned to a variable that would possibly live longer than the object itself.

When a constructor has this attribute, it means that the newly constructed object may reside within a `ScopedMemory` context. If a constructor is declared to comply with the `@ScopedThis` annotation, the constructor's initialization code executes in the same dynamic scope that represents the newly constructed object. Thus, any objects allocated within the constructor will reside in the same scope as `this`. On the other hand, if a constructor does not have the `@ScopedThis` annotation, a new scope is created for the allocation of temporary objects within the constructor.

### `@Scoped`

(applies to instance and static fields, methods, method parameters, and local variables.)

When applied to an instance field, `@Scoped` signals the intention to create linked data structures out of stack-allocated objects. The contents of a `@Scoped` field can only be copied to other variables that are also declared to have the `@Scoped` attribute. In the most general case, writing to a `@Scoped` field requires a run-time check to ensure that if the value to be written refers to stack-allocated memory, the object that contains the `@Scoped` variable resides on the same stack in a more inner scope than the referenced object. In the special case that the object containing the `@Scoped` instance field was just allocated in the inner-most active memory scope, the run-time check shall be avoided.

Consider a scenario in which certain real-time classes are dynamically loaded into an outer-nested scope for use only by other components that are more inner-nested on the scope stack. In this scenario, it makes sense to declare that certain static variables associated with this class have the `@Scoped` attribute. The contents of any such variable can only be copied to other variables that are also declared to have the `@Scoped` attribute. In the most general case, writing to a `@Scoped` static field requires a run-time check to ensure that if the value to be written refers to stack-allocated memory, the object that contains the `@Scoped` variable resides on the same stack in a more inner

scope than the referenced object. In the special case that the object containing the `@Scoped` static variable was just allocated in the inner-most active memory scope, the run-time check shall be avoided.

When applied to a method, this annotation denotes that the method may return a reference to an object that resides in scoped memory. In the most general case, a run-time check is required before returning from the method to assure that the value to be returned resides within a scope that is nested outside the scope of the returning method. In the special case that a reachability data-flow analysis within the method demonstrates that the return value was not allocated within this method or any method it calls, the run-time check shall be avoided. The byte-code verifier assures that any invocation of a method that has this annotation shall assign the result of the method to a variable that has the `@Scoped` attribute.

When applied to a parameter variable, the `@Scoped` attribute means the variable's contents can only be assigned to other local or parameter variables that are also declared with the `@Scoped` annotation, or to instance and static fields that are declared with the `@Scoped` annotation after performing the appropriate run-time checks.

The `@Scoped` annotation may also be applied to local variables for code documentation purposes. However, these annotations are not preserved in class files, so they do not directly affect byte-code verification or code generation. By default, every local reference variable is considered to have the `@Scoped` attribute. The byte-code verifier removes this attribute from a local variable only if its analysis of the code reveals that (a) this variable's contents is copied to an outgoing argument which was not declared to have the `@Scoped` attribute, (b) this variable's contents is copied to an instance or static variable that does not have the `@Scoped` attribute, (c) this variable's contents is copied to another local or parameter variable that does not have the `@Scoped` attribute, or (d) this variable's contents is returned from this method, and the method does not have the `@Scoped` attribute.

Every `new()` operation that assigns its result directly to a variable or outgoing parameter with the `@Scoped` attribute shall allocate its memory within the scope of the current activation frame. All other allocations shall use `ImmortalMemory`.

### `@ScopedLocal`

(applies to arguments of methods declared with `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` annotations and constructors with `@ScopedThis` annotations)

This annotation indicates that the incoming argument refers to scoped memory which will not be referenced from the caller-allocated result object if the annotation accompanies a method declaration, or from the constructed object if the annotation accompanies a constructor. Use this annotation when it is desirable to allow `@Scoped` arguments that potentially reside in more inner-nested scopes than the scope that will ultimately hold the object to be allocated by the invoked method or constructor.

The byte-code verifier assures that `@ScopedLocal` values and values copied from within an object referenced from a `@ScopedLocal` variable are never copied to other variables that are not identified as `@ScopedLocal` variables.

**@ScopedArrayLocal**

(applies to arguments of methods declared with **@CallerAllocatedResult** or **@CallerAllocatedArrayResult** annotations and constructors with **@ScopedThis** annotations)

This annotation indicates that the incoming argument is an array of references which might reside in scoped memory, and each element of which might refer to an object residing in scoped memory. The annotation further means neither this array, nor the elements of this array will be referenced from the caller-allocated result object if the annotation accompanies a method declaration, or from the constructed object if the annotation accompanies a constructor. Use this annotation when it is desirable to allow **@Scoped** arguments that potentially reside in more inner-nested scopes than the scope that will ultimately hold the object to be allocated by the invoked method or constructor.

The byte-code verifier assures that **@ScopedArrayLocal** values are never copied to other variables that are not identified as **@ScopedArrayLocal** variables, and that the elements of the array are not copied to variables that are not identified as **@ScopedLocal** variables.

**@ScopedThisLocal**

(applies to methods declared with **@CallerAllocatedResult** or **@CallerAllocatedArrayResult** annotations)

This annotation indicates that the value of **this** within the annotated method refers to scoped memory which will not be referenced from the caller-allocated result of this method. Use this annotation when it is desirable to allow an implicit **@ScopedThis** argument that potentially resides in more inner-nested scopes than the scope that will ultimately hold the object to be allocated by the invoked method or constructor.

The byte-code verifier assures that the **@ScopedThisLocal** value and values copied from within the object referenced from the **@ScopedThisLocal** variable are never copied to other variables that are not identified as **@ScopedLocal** variables.

**@ScopedArray**

(applies to instance and static variables, parameters, and methods)

This annotation applies only to entities that represent arrays of references. In general, this attribute denotes that the array's elements may themselves refer to objects residing in scoped memory. A **@ScopedArray** variable may only be assigned to other **@ScopedArray** variables. An element of a **@ScopedArray** variable may only be assigned to **@Scoped** variables. Assignments to an element of a **@ScopedArray** object may require a run-time check, depending on the context.

When applied to a method, the **@ScopedArray** annotation denotes that the result returned from the method is an array that may reside in scoped memory, and each entry within the array may also reside in scoped memory.

**@ScopedPure**

(applies to methods and constructors)

This annotation is short-hand to denote that all reference arguments, including **this**, have the **@Scoped** attribute. If any of the arguments refers to an array of references, that argument is considered to have the **@ScopedArray** attribute. **@ScopedPure** further implies that this method, and

any method that overrides it, does not have the `@AllowCheckedScopedLinks` attribute. `@ScopedPure` does not imply that the method's result is `@Scoped` or `@ScopedArray`. Neither does it imply that the result is caller allocated (See `@CallerAllocatedResult` and `@CallerAllocatedArrayResult`).

**@CallerAllocatedResult** (subclasses = { <list of sub-class types> })

This method annotation indicates that the `Object` to be returned from this method may be allocated within the allocation context of the caller. Besides placing certain restrictions on the body of this method, the presence of this annotation requires that the calling method set aside sufficient memory to hold the method's return result and pass the address of this allocation buffer implicitly to the method. The optional list of sub-classes specifies the allowed sub-class types that might be returned from this method. The caller must set aside sufficient memory to represent the largest of these sub-class representations.

**@CallerAllocatedArrayResult** (subclasses = { list of sub-class types })

This method annotation may be used to annotate a method that returns an array of references. This annotation indicates that the array object to be returned from this method, along with all of the objects that are directly referenced by this array, may be allocated within the caller's stack activation frame. Besides placing certain restrictions on the body of this method, the presence of this annotation requires that the calling method set aside sufficient memory to hold the reference array to be returned from this method plus enough additional memory to represent each of the objects to be referenced from the returned reference array. The caller passes the address of this allocation buffer implicitly to the method. The optional list of sub-classes specifies the allowed sub-class types that might be assigned to individual array elements. The caller must set aside sufficient memory to represent the worst-case in which every array element is the largest of the enumerated sub-class types.

**@ScopedMemorySize** (bytes = <N>)

(applies to methods and constructors)

For methods that are not fully analyzable, this annotation allows programmers to specify the desired size of the method's `ScopedMemory` allocation context, measured in bytes. If a method's body is fully or partially analyzable, the size of the method's allocation context will be determined analytically for all modes that are analyzable. For modes that are not analyzable, the size of the scope will be determined by this annotation. See the documentation for this annotation on page page 178. There are various alternative mechanisms available for characterizing the desired scope size.

**@AllowCheckedScopedLinks**

(applies to methods and constructors)

By default, the byte-code verifier rejects any code that would require a run-time check in order to assure integrity of references between objects allocated in `ScopedMemory` regions. There is a class of algorithms, however, for which static analysis of the code cannot prove that all assignments will be legal. If developers find it necessary to use these algorithms, they must annotate any method that might contain an assignment operation that needs to be checked with this annotation. With this annotation present, the byte-code verifier will allow checked assignment operations. To further draw attention to the risks associated with inclusion of code that might include an illegal assignment operation, the byte-code verifier requires that any method with this annotation is

declared to throw `javax.realtime.util.mc.IllegalAssignmentException`. Note that this is different than `IllegalAssignmentError`, which is an unchecked exception.

#### `@ImmortalAllocation`

(applies to methods and constructors)

This annotation indicates the method may allocate `ImmortalMemory`. A method that may allocate `ImmortalMemory` can only be called by other methods that may allocate `ImmortalMemory`. The byte-code verifier rejects any methods that might allocate `ImmortalMemory` if the method is not annotated with the `@ImmortalAllocation` annotation. Unlike most of the other memory-scope annotations, this particular attribute is not inherited to overriding methods in sub-classes.

## 7. Using Nested Scopes for Composition of Modular Software Systems

The proposed dynamic memory allocation abstractions for the safety-critical and hard real-time mission-critical RTSJ profiles require that memory for particular modules be allocated and deallocated in LIFO order, with certain modules being contained entirely within other modules. In general, higher layer soft real-time software is much more dynamic and lower level hard real-time software tends to be much more static. However, even the hard real-time components occasionally need to be able to reconfigure their use of memory.

There is design tension between the need to support very efficient and reliable operation, and the desire to support flexibility and dynamic reconfiguration in the field. Many mission-critical systems are required to support non-stop operation. It may be difficult or cost prohibitive to shut down the system each time software reconfiguration is required. Consider, for example, some common needs for flexibility in field-deployed systems:

- An error is discovered in a network protocol stack and a new version of this software must be installed to replace the faulty module.
- An orbiting communications satellite is required to support new services that were not anticipated or fully standardized when the satellite was originally launched. This likely requires the installation of new control-plane and management-plane functionality.
- A hardware failure in a line card of a highly available telecommunication server requires that the line card be hot-swapped with a new replacement. Since the replacement hardware has a different ROM revision number than the original hardware, certain shelf controller device drivers must be updated.
- A diagnostic computer is hooked up to a running telecommunication switch. The diagnostic system installs certain temporary software components into the telecommunication switch in order to gather the required diagnostic information.
- A sensor in a rocket ship fails. If the sensor has only partially failed, it may be possible to remedy this situation by replacing the sensor's device driver with a new driver that simply filters and recalibrates sensor readings. In other cases, it may be possible to replace the failed sensor's device driver with a software module that approximates the desired sensor data by assimilating information from alternative sources. For example, if one of four redundant sensors fails, the failed sensor readings might be replaced with the average reading from the other three.
- Due to limited communication bandwidth between UAV (Unmanned aerial vehicle) or orbiting satellite or deep-space probe or ASW (Anti-Submarine Warfare) buoy and "central control", certain assimilation and fusion of sensor data must be performed by remotely deployed devices. There is simply not

**(Permission granted to reproduce and distribute as a complete document, without modification)**

enough bandwidth to transmit all gathered sensor information back to headquarters. In some cases, such as with deep-space probes, the long round-trip communication delay precludes the use of closed loop feedback-control algorithms that include earth involvement in the closed loop. In all of these cases, it may be necessary to upload custom-tailored information processing or control loop algorithms into the remotely deployed systems in order to achieve maximum utilization of the limited communication bandwidth based on specific needs and circumstances.

- A communication engineer plugs a new sensor into a software-controlled field radio so that new sensor information can be transmitted to the command and control center. This new sensor, which likely requires installation of certain device driver software into the intelligent radio system, might provide seismic, sonar, radar, visual, or laser-based weapons targeting data.

Within the soft real-time domain, a real-time implementation of J2SE provides excellent support for dynamic reconfiguration of software. New classes can be loaded. Obsolete classes can be unloaded. Defragmenting real-time garbage collection efficiently and reliably manages the provisioning of memory resources to support the newly installed functionality.

It is more difficult to maintain compliance with high performance and hard real-time constraints in the face of software reconfiguration. System architects must carefully organize their hard real-time architecture so as to facilitate modular replacement and enhancement of particular software components within the architecture.

The first step is to minimize dependencies between individual components. Consider, for example, an idealized architecture for an aircraft mission control system. This particular example is contrived to illustrate the mixing of hard and soft real-time components, and of components that are considered safety-critical with those that are more in the realm of the mission-critical domain. In a real system, there would be stronger partitioning and stronger isolation of concerns between these hypothetical components. The necessary modules might consist of the following:

1. Interface to global positioning hardware
2. Interface to air traffic control communication subsystem
3. Interface to meteorological communication
4. Interface to aircraft sensors (fuel gauges, temperature, pressure, wind speed)
5. Module to provide graphical user interface
6. Module to maintain map data base
7. Module to maintain and assimilate relevant air traffic control information
8. Module to maintain and assimilate relevant meteorological information
9. Module to maintain current flight plan
10. Module to track flight progress against current plan
11. Module to support what-if scenario analysis (Do I have sufficient fuel to divert to alternative destination?)
12. Module to support revision to or replacement of the current flight plan
13. Module to perform auto-pilot functions

**(Permission granted to reproduce and distribute as a complete document, without modification)**

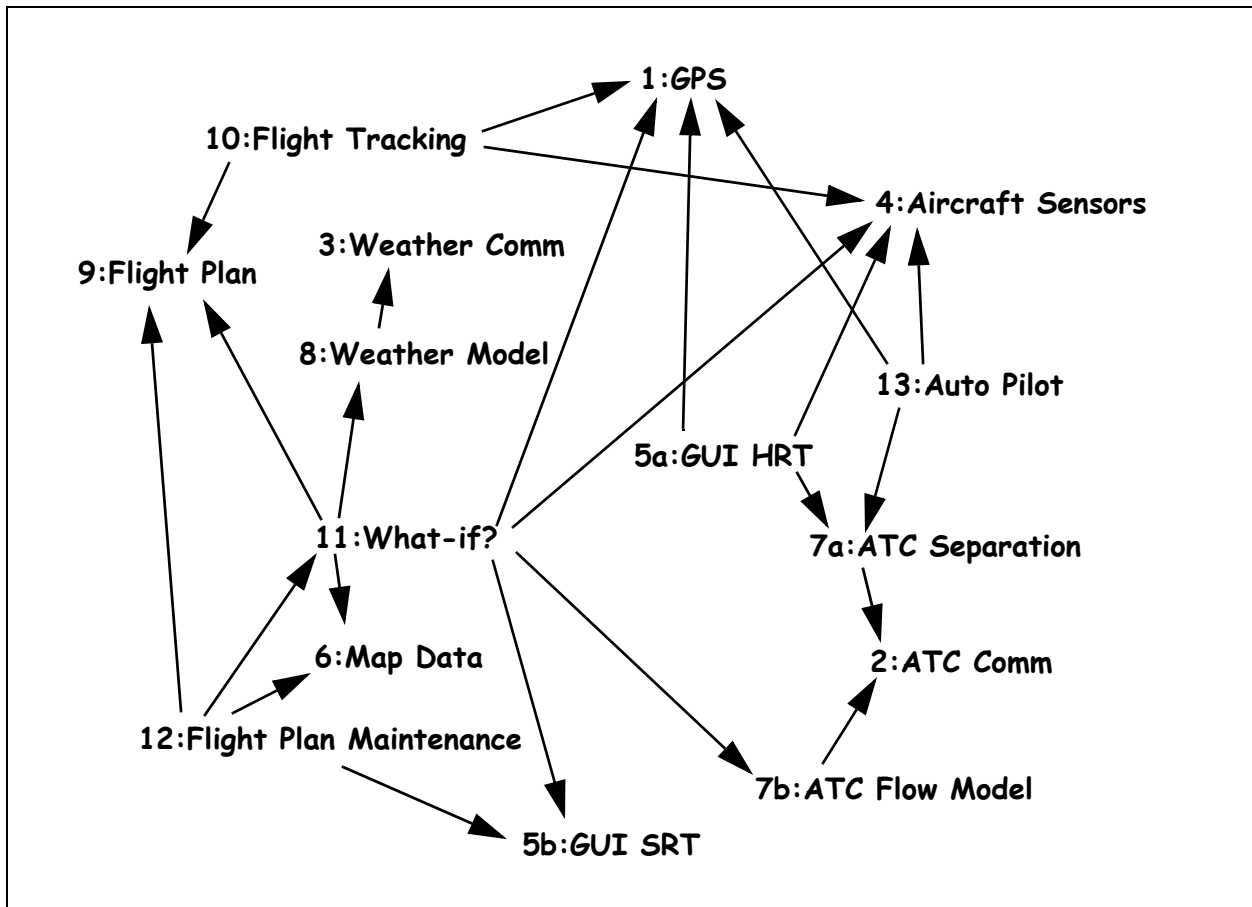
In considering how to architect this particular software system, the architect is likely to begin by characterizing the criticality and timeliness requirements associated with each of the components in the system. Suppose the architect’s analysis characterizes the individual components according to the chart illustrated in Figure 8. Note that we have divided the Graphical User Interface (component 5) into two components,

<b>Hard Real Time</b>	<b>9:Flight Plan</b>  <b>10:Flight Tracking</b>  <b>3:Weather Comm</b>	<b>1:GPS</b>  <b>4:Aircraft Sensors</b>  <b>2:ATC Comm 5a:GUI HRT</b>  <b>7a:ATC Separation Control</b>  <b>13:Auto Pilot</b>
	<b>Soft Real Time</b>	<b>7b:ATC Flow Model</b>  <b>8:Weather Model</b>  <b>6:Map Data      5b:GUI SRT</b>  <b>11:What-if? Analysis</b>  <b>12:Flight Plan Maintenance</b>
	<b>Mission Critical (DO-178B Level C)</b>	<b>Safety Critical (DO-178B Level A)</b>

Figure 8: Criticality and Timeliness Requirements for Architectural Components

one representing the safety-critical glass-cockpit information display that is required to manually fly the airplane, and the other representing more general purpose interactive display for use by a flight or navigation engineer who might be responsible for making changes to the planned mission. Note also that the stream of digital data represented by the Air Traffic Control radio is communicated both to the airplane minimum separation enforcement module (7a) and to the flow control module (7b). Air Traffic Flow Control is provisioned in this model as a mission-critical function.

For purposes of this discussion, assume that all of the software is to be implemented on the same microprocessor running a partitioned ARINC-653 kernel. The six level-A components are all implemented within one partition and all of the level-C components are implemented in a different partition. Figure 9 illustrates the dependencies between the various components. An arrow from one component to another indicates that the first depends on the second. Note that the dependency analyses in these figures describe software maintenance dependencies. When we say, for example, that the “Flight Tracking” module depends on the “Flight Plan Interface”, we are saying that any changes to the “Flight Plan Interface” may require changes to the “Flight Tracking” module. Ultimately, this represents our best guess as to future software maintenance requirements. In general, we are more confident that we can hold interfaces stable while evolving the implementations of certain modules that communicate with each other by way of these interfaces.



**Figure 9: Inherent Dependency Analysis Between Components**

The dependency analysis illustrated in Figure 9 represents inherent source-code maintenance dependencies. However, it ignores partitioning, flexibility, and maintainability issues. To address these issues, it is necessary to introduce standard interfaces and shared buffers into the dependency model. A dependency analysis of the enhanced model is illustrated in Figure 10.

In Figure 10, we have color coded the objects to emphasize their distinct realms of responsibility. All of the components residing in the safety-critical domain are colored red. Blue objects reside in the hard real-time mission-critical domain, and black objects reside in the soft real-time (traditional Java) domain. We use green to represent ARINC 653 ports that connect safety-critical to mission-critical components. Further, we assume that only hard real-time components can communicate directly with these ARINC 653 ports. Thus, whenever a traditional Java component requires access to ARINC 653 port data, an intermediary hard real-time component serves to shuttle the data between the respective domains.

Given this organization of software components, we would first divide the software between safety-critical and mission-critical functionality. Then, we would carefully organize the memory partitions for each of the two hard real-time execution domains. The safety-critical partition might be arranged as shown in Figure 11. The suggested organization of the mission-critical partition is shown in Figure 12.

Note in Figure 11 that there are three layers of nested scopes. The level-0, or outermost scope, holds the internal interfaces to key components. These interfaces are placed in the outer-most scope to make them visible to all of the inner-nested scopes that represent the individual components in the system. Scope lev-

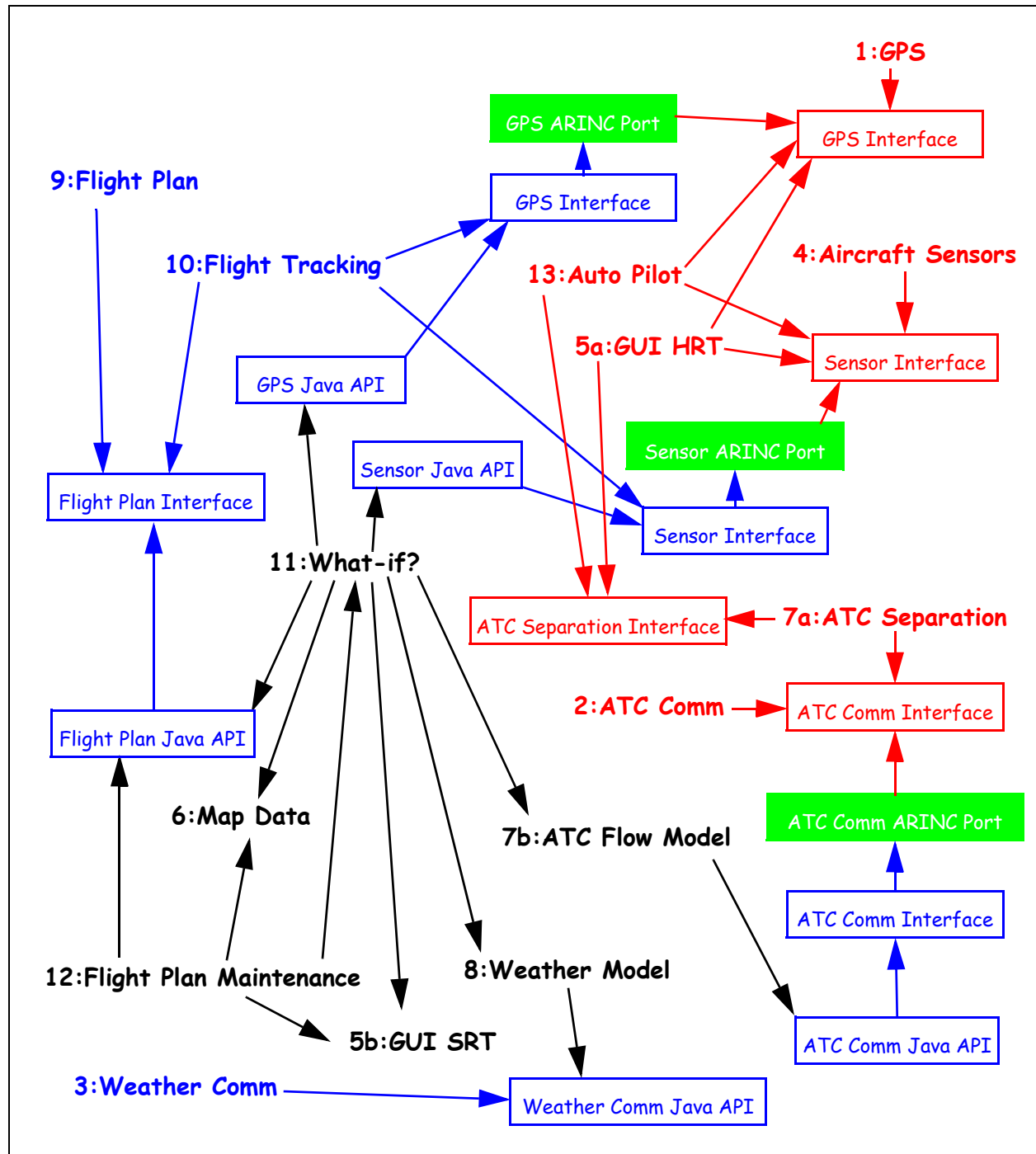


Figure 10: Dependency Analysis of Modules and Interfaces

els 0 and 1 both hold collections of objects. In the third scope level, we allocate multiple independent scopes, one to represent each of the threads that comprises the functional modules of the safety-critical partition.

The memory organization illustrated in Figure 12 suggests the use of four nested scope levels within the mission-critical partition. Java code to initialize the mission-critical partition is provided in Figures 13 through 18. Figure 13 provides the code to initialize the *Level0* data structures. Note that the four variables

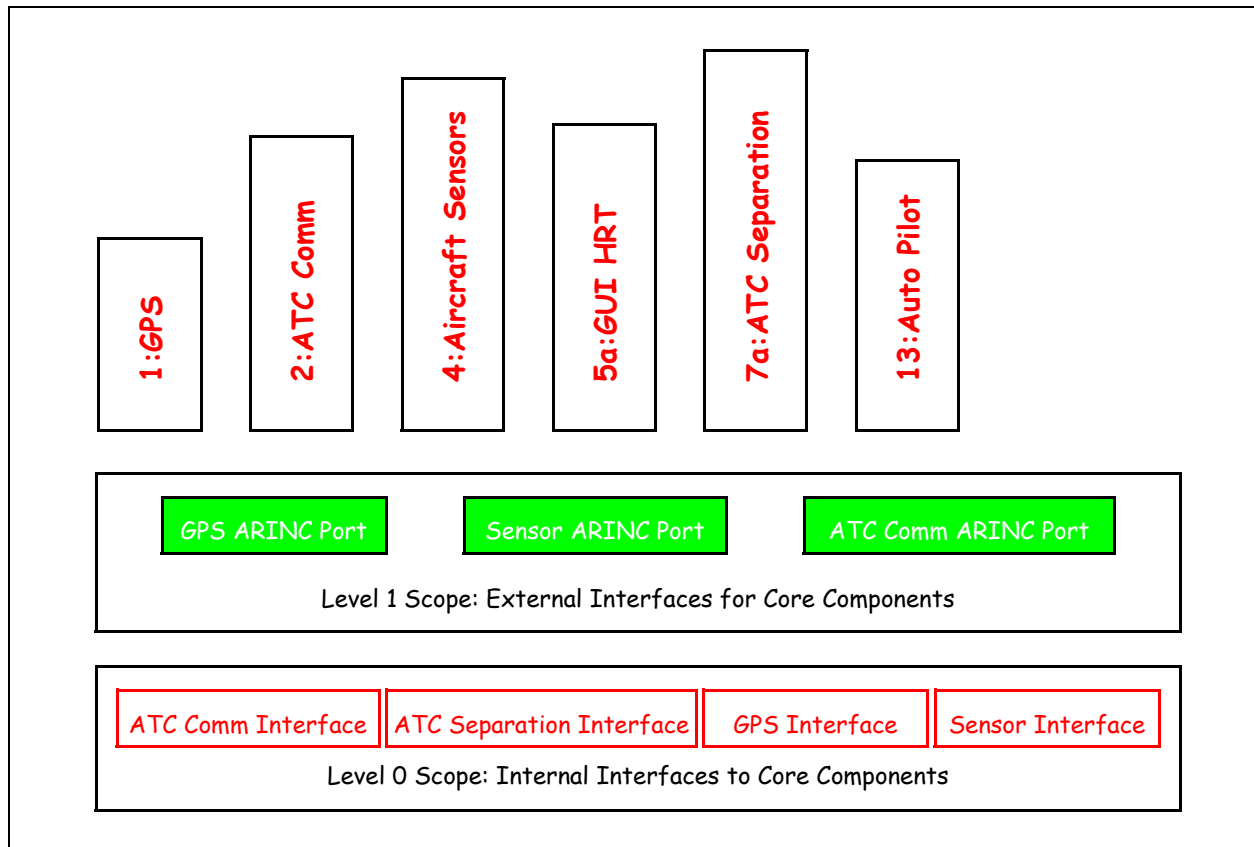


Figure 11: Organization of Safety-Critical Memory Partition

`atc`, `sensor`, `gps`, and `fp` are all declared with the `@Scoped` attribute. This indicates the programmer's intent that these variables may refer to objects allocated within the same scope as this `Level0` object or in more outer-nested scopes. All of the assignments to these variables are made within this object's constructor. Thus, no run-time assignment checks are required when these variables are initialized since all of the objects are allocated within the same scope.

The `inner_scope` variable represents all of the memory dedicated to inner-nested scopes. The RTSJ rules of referential integrity prohibit the `Level0` scope from holding any references to objects allocated within the inner-nested scopes. `ThreadStack` is a proposed new mission-critical abstraction designed to make possible the creation of scopes within scopes. The `ThreadStack` object itself is allocated within the `Level0` scope. But any objects allocated within the `ThreadStack`, including the thread that is spawned on line 25 of Figure 13, are allocated within inner-nested scopes and cannot be seen from within the `Level0` object.

The proposed rules for use of the `ThreadStack` abstraction within mission-critical code require that each invocation of the `spawn()` method be paired with a `finally` invocation of the same `ThreadStack` object's `join()` method, as shown, for example, on lines 23 through 28 of Figure 13 and lines 33 through 47 of Figure 14. This is to be enforced by the byte-code verifier.

The arguments to the `spawn()` invocation are:

1. A `@Scoped Class` object which must derive from `NoHeapRealtimeThread`. The `spawn()` method assures that this `Class` argument derives from `NoHeapRealtimeThread`, and further requires that the

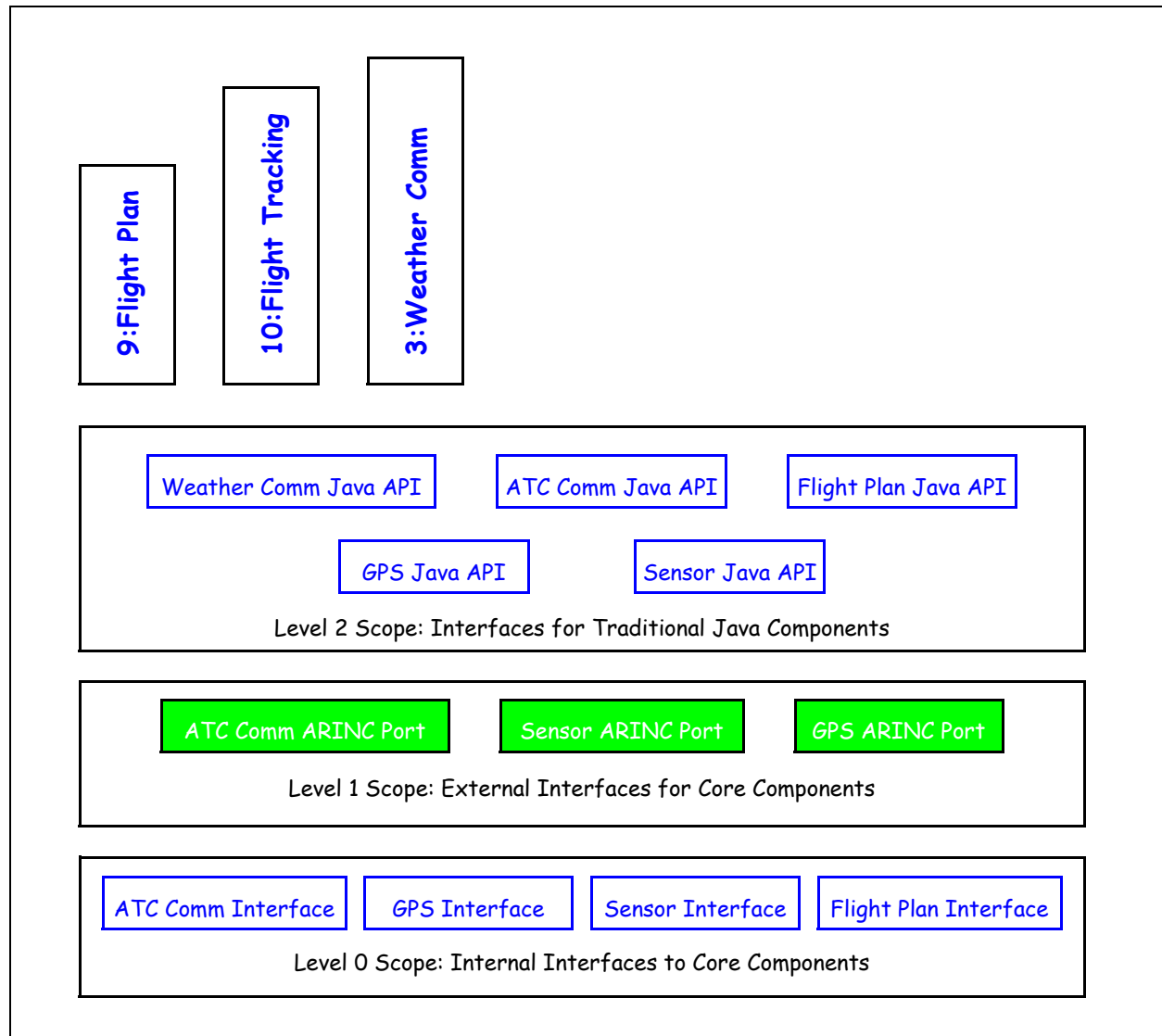


Figure 12: Organization of Hard Real-Time Mission-Critical Partition

class has a constructor that takes as arguments instances of `Object`, `SchedulingParameters`, and `MemoryArea`.

2. A `@Scoped Object` which will be passed as an argument to the constructor of the newly spawned thread. In this case, we are passing a reference to the `Level0` object itself, as this makes it possible for inner-nested scopes to gain access to the shared `atc`, `sensor`, `gps`, `fp`, and `inner_scope` variables.
3. A `@Scoped SchedulingParameters` value which governs the scheduling of the newly spawned thread.

The three parameters are all declared to have the `@Scoped` attribute because the spawned thread promises to keep these references within its scope and not copy them into global locations. Another form of the `spawn()` method omits the second (`Object`) argument. Use this second form to instantiate the primordial `Level0` object.

Figure 14 provides initialization of the the `Level1` class. The `Level1` constructor is automatically invoked from within the `spawn()` invocation on line 25 of Figure 13. Note that the first argument of the constructor

(Permission granted to reproduce and distribute as a complete document, without modification)

```

[1] public class Level0 extends NoHeapRealtimeThread {
[2]     private final static int SIZE_OF_INNER_SCOPES = 4096 * 9;
[3]     private final static int NESTED_PRIORITY_LEVEL = 18;
[4]
[5]     @Scoped ATC_CommInterface atc;
[6]     @Scoped SensorInterface sensor;
[7]     @Scoped GPS_Interface gps;
[8]     @Scoped FlightPlanInterface fp;
[9]     @Scoped ThreadStack inner_scope;
[10]
[11]     public @ScopedPure Level0(SchedulingParameters parms, MemoryArea area) {
[12]         super(parms, area);
[13]         atc = new ATC_CommInterface();
[14]         sensor = new SensorInterface();
[15]         gps = new GPS_Interface();
[16]         fp = new FlightPlanInterface();
[17]         inner_scope = new ThreadStack(SIZE_OF_INNER_SCOPES);
[18]     }
[19]
[20]     public @ScopedThis void run() {
[21]         PriorityParameters parms;
[22]
[23]         try {
[24]             parms = new PriorityParameters(NESTED_PRIORITY_LEVEL);
[25]             inner_scope.spawn(Level1.class, this, parms);
[26]         } finally {
[27]             inner_scope.join();
[28]         }
[29]     }
[30] }

```

Figure 13: Code to Initialize the **Level0** Class

is copied from the second argument to `spawn()`. We coerce this argument to a `Level0` reference, thereby obtaining access to the various outer-nested objects that are used by the Level-1 objects instantiated within this constructor. Note in line 22, for example, that construction of the level-1 `GPS_ARINC_PORT` object takes as an argument a reference to the level-0 `GPS_Interface` object that is represented by the `gps` instance variable within the `Level0` object.

Since each of the instance variables initialized by the constructor is annotated with the `@Scoped` attribute, all of the objects constructed within this constructor are allocated within the same `ThreadStack` memory scope that holds this `Level1` object. This means all of the memory for all of these objects can be instantly reclaimed when the `Level1 run()` method terminates. If any of these instance variables had not been labeled with the `@Scoped` annotation, the corresponding constructor would have been forced to allocate the memory for that object out of `ImmortalMemory` rather than using the temporary memory represented by the enclosing `ThreadStack` context.

The `run()` method is shown in Figure 16. Note the use of the `@TraditionalJavaShared` annotation on line 1. This signifies that the objects allocated within this method's private scope may be shared with traditional Java components. Each of the `javax.jmc.Registry.publish()` invocations (shown on lines 31 through 35) checks to ensure that the scope containing the published object is declared with the `@TraditionalJav-`

```
[1] public class Level1 extends NoHeapRealtimeThread {
[2]     private final static int THREAD_STACK_SIZE = 4096;
[3]     private final static int SIZE_OF_INNER_SCOPES = 4096 * 4;
[4]     private final static int NESTED_PRIORITY_LEVEL = 18;
[5]     private final static int GPS_PRIORITY_LEVEL = 30;
[6]     private final static int ATC_PRIORITY_LEVEL = 24;
[7]     private final static int SENSOR_PRIORITY_LEVEL = 24;
[8]
[9]     @Scoped GPS_ARINC_port gps_port;
[10]    @Scoped ATC_ARINC_port atc_port;
[11]    @Scoped Sensor_ARINC_port sensor_port;
[12]    @Scoped ThreadStack gps_memory;
[13]    @Scoped ThreadStack atc_memory;
[14]    @Scoped ThreadStack sensor_memory;
[15]    @Scoped ThreadStack inner_scope;
[16]    @Scoped Level0 outer_context;
[17]
[18]    public @ScopedPure Level1(Object outer_object,
[19]                             SchedulingParameters parms, MemoryArea area) {
[20]        super(parms, area);
[21]        outer_context = (Level0) outer_object;
[22]        gps_port = new GPS_ARINC_port(outer_context.gps);
[23]        atc_port = new ATC_ARINC_port(outer_context.atc);
[24]        sensor_port = new Sensor_ARINC_port(outer_context.sensor);
[25]        gps_memory = new ThreadStack(THREAD_STACK_SIZE);
[26]        atc_memory = new ThreadStack(THREAD_STACK_SIZE);
[27]        sensor_memory = new ThreadStack(THREAD_STACK_SIZE);
[28]        inner_scope = new ThreadStack(SIZE_OF_INNER_SCOPES);
[29]    }
[30]
[31]    public @ScopedThis void run() {
[32]        PriorityParameters parms;
[33]        try {
[34]            parms = new PriorityParameters(GPS_PRIORITY_LEVEL);
[35]            gps_memory.spawn(GPS_ARINC_thread.class, gps_port, parms);
[36]            parms = new PriorityParameters(ATC_PRIORITY_LEVEL);
[37]            atc_memory.spawn(ATC_ARINC_thread.class, atc_port, parms);
[38]            parms = new PriorityParameters(SENSOR_PRIORITY_LEVEL);
[39]            sensor_memory.spawn(Sensor_ARINC_thread.class, sensor_port, parms);
[40]            parms = new PriorityParameters(NESTED_PRIORITY_LEVEL);
[41]            inner_scope.spawn(Level2.class, this, parms);
[42]        } finally {
[43]            inner_scope.join();
[44]            gps_memory.join();
[45]            atc_memory.join();
[46]            sensor_memory.join();
[47]        }
[48]    }
[49] }
```

Figure 14: Code to Initialize the **Level1** Class

```

[1] public class Level2 extends NoHeapRealtimeThread {
[2]     private final static int THREAD_STACK_SIZE = 4096;
[3]     private final static int FPT_PRIORITY_LEVEL = 12;
[4]     private final static int FTT_PRIORITY_LEVEL = 24;
[5]     private final static int WCT_PRIORITY_LEVEL = 4;
[6]
[7]     private static final int SHUTDOWN = 0;
[8]     private static final int REPLACE_FLIGHT_PLAN_MODULE = 1;
[9]     private static final int REPLACE_FLIGHT_TRACKING_MODULE = 2;
[10]    private static final int REPLACE_WEATHER_COMM_MODULE = 3;
[11]    private static final int IDLE = 4;
[12]
[13]    @Scoped ThreadStack fpt_memory, ftt_memory, wct_memory;
[14]
[15]    @Scoped Level1 outer_context;
[16]
[17]    private int request = IDLE;
[18]
[19]    @Scoped Class replace_class;
[20]    @Scoped Object replace_arg;
[21]    @Scoped SchedulingParameters replace_parms;
[22]
[23]    public @ScopedPure Level2(Object outer_object,
[24]                               SchedulingParameters parms, MemoryArea area) {
[25]        super(parms, area);
[26]        outer_context = (Level1) outer_object;
[27]
[28]        fpt_memory = new ThreadStack(THREAD_STACK_SIZE);
[29]        ftt_memory = new ThreadStack(THREAD_STACK_SIZE);
[30]        wct_memory = new ThreadStack(THREAD_STACK_SIZE);
[31]    }
[32]

```

**Figure 15: Variable Declarations and Constructor for Level2 Class**

`aShared` attribute. If not, it throws an `IllegalArgumentException` and refuses to expose the object to the traditional Java domain. Further, the byte-code verifier assures that any method that is declared with the `@TraditionalJavaShared` attribute includes an invocation of `javax.jmc.Registry.awaitClearRegistry()` as the last statement in the `finally` block of code associated with the `try` statement that surrounds the body of code that comprises this method's body. See line 47. The `awaitClearRegistry()` invocation blocks the current thread until the traditional Java environment no longer holds any proxies for objects residing in this method's private scope.

The `Level2` class introduces the ability to selectively replace certain mission-critical program modules without shutting down the entire system. This somewhat simplistic code illustrates the flexibility and capabilities of the proposed mission-critical composition hierarchy. It does not necessarily represent the ideal architecture for dynamic reconfiguration of mission-critical software components.

The `doMaintenance()` method is shown in Figure 17. This has the responsibility for dynamically reconfiguring certain Level-2 modules within a running mission-critical system. The `request` instance variable is set separately to request that a particular module be replaced (See Figure 18). The `joinAndSpawn()` method

```
[1] public @TraditionalJavaShared @ScopedThis void run() {
[2]     @Scoped Java_Weather_COMM weather = null;
[3]     @Scoped Java_ATC_COMM atc = null;
[4]     @Scoped Java_FlightPlan fp = null;
[5]     @Scoped Java_GPS gps = null;
[6]     @Scoped Java_SensorInterface sensor = null;
[7]     Object [] flight_tracking_args;
[8]     PriorityParameters parms;
[9]     javax.jmc.Registry registry = javax.jmc.Registry.instance();
[10]    try {
[11]        weather = new Java_Weather_COMM();
[12]        atc = new Java_ATC_COMM(this.outer_context.outer_context.atc);
[13]        fp = new Java_FlightPlan(this.outer_context.outer_context.fp);
[14]        gps = new Java_GPS(this.outer_context.outer_context.gps);
[15]        sensor = new Java_SensorInterface(this.outer_context.outer_context.sensor);
[16]
[17]        parms = new PriorityParameters(FPT_PRIORITY_LEVEL);
[18]        fpt_memory.spawn(FlightPlanThread.class,
[19]                        outer_context.outer_context.fp, parms);
[20]
[21]        parms = new PriorityParameters(FTT_PRIORITY_LEVEL);
[22]        flight_tracking_args = new Object[3];
[23]        flight_tracking_args[0] = outer_context.outer_context.fp;
[24]        flight_tracking_args[1] = outer_context.outer_context.gps;
[25]        flight_tracking_args[2] = outer_context.outer_context.sensor;
[26]        ftt_memory.spawn(FlightTrackingThread.class, flight_tracking_args, parms);
[27]
[28]        parms = new PriorityParameters(WCT_PRIORITY_LEVEL);
[29]        wct_memory.spawn(WeatherCommThread.class, weather, parms);
[30]
[31]        registry.publish("Weather Data", weather);
[32]        registry.publish("ATC Flow Control", atc);
[33]        registry.publish("Flight Plan", fp);
[34]        registry.publish("Global Positioning", gps);
[35]        registry.publish("Sensor Data", sensor);
[36]
[37]        this.doMaintenance();
[38]    } finally {
[39]        registry.unpublish(weather);
[40]        registry.unpublish(atc);
[41]        registry.unpublish(fp);
[42]        registry.unpublish(gps);
[43]        registry.unpublish(sensor);
[44]        fpt_memory.join();
[45]        ftt_memory.join();
[46]        wct_memory.join();
[47]        Registry.instance().awaitClearRegistry();
[48]    }
```

Figure 16: The run() Method for Level2 Class

```
[1]  @ScopedThis synchronized void doMaintenance() {
[2]      while (request != SHUTDOWN) {
[3]          switch (request) {
[4]
[5]              case REPLACE_FLIGHT_PLAN_MODULE: {
[6]                  fpt_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[7]                  request = IDLE;
[8]                  this.notifyAll();
[9]                  break;
[10]             }
[11]
[12]             case REPLACE_FLIGHT_TRACKING_MODULE: {
[13]                 ftt_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[14]                 request = IDLE;
[15]                 this.notifyAll();
[16]                 break;
[17]             }
[18]
[19]             case REPLACE_WEATHER_COMM_MODULE: {
[20]                 wct_memory.joinAndSpawn(replace_class, replace_arg, replace_parms);
[21]                 request = IDLE;
[22]                 this.notifyAll();
[23]                 break;
[24]             }
[25]
[26]             case IDLE:
[27]             default:
[28]
[29]             }
[30]             this.wait();
[31]         }
[32]     }
```

Figure 17: The `doMaintenance()` method for `Level2` Class

first waits for the originally spawned thread to terminate and then instantiates a new thread in its place, overwriting the memory that had previously been reserved for the first thread. This illustration does not show the code that arranges for the original thread to terminate its execution. That must be handled elsewhere.

Remember that the byte-code verifier assures that every `spawn()` invocation is accompanied by a `join()` within an accompanying `finally` block of code. It is important to maintain this invariant in order to assure that program control does not leave any context before all of the references to that context have been abandoned. In preserving this same invariant, the `joinAndSpawn()` method enforces the following:

- This `ThreadStack` must have successfully spawned a thread which has not yet been joined by the parent thread. Otherwise, `joinAndSpawn()` does not wait to join the subordinate thread and instead throws `IllegalStateException`.
- Any `@Scoped` variables passed as arguments to the `joinAndSpawn()` method must belong to scopes that belong to the same scope or to scopes more outer-nested than the scope that hosted the original `spawn()` invocation. This is necessary because that scope contains the `join()` operation that preserves

```
[1]  @AllowCheckedScopedLinks @ScopedThis synchronized void
[2]      changeWeatherCommModule(@Scoped Class new_class,
[3]                              @Scoped Object new_arg,
[4]                              @Scoped SchedulingParameters new_parms) {
[5]      while (request != IDLE)
[6]          this.wait();
[7]      replace_class = new_class;
[8]      replace_arg = new_arg;
[9]      replace_parms = new_parms;
[10]     request = REPLACE_WEATHER_COMM_MODULE;
[11]     this.notifyAll();
[12] }
[13]
[14] @AllowCheckedScopedLinks @ScopedThis synchronized void
[15]     changeFlightTrackingModule(@Scoped Class new_class,
[16]                               @Scoped Object new_arg,
[17]                               @Scoped SchedulingParameters new_parms) {
[18]     while (request != IDLE)
[19]         this.wait();
[20]     replace_class = new_class;
[21]     replace_arg = new_arg;
[22]     replace_parms = new_parms;
[23]     request = REPLACE_FLIGHT_TRACKING_MODULE;
[24]     this.notifyAll();
[25] }
[26]
[27] @AllowCheckedScopedLinks @ScopedThis synchronized void
[28]     changeFlightPlanModule(@Scoped Class new_class,
[29]                            @Scoped Object new_arg,
[30]                            @Scoped SchedulingParameters new_parms) {
[31]     while (request != IDLE)
[32]         this.wait();
[33]     replace_class = new_class;
[34]     replace_arg = new_arg;
[35]     replace_parms = new_parms;
[36]     request = REPLACE_FLIGHT_PLAN_MODULE;
[37]     this.notifyAll();
[38] }
[39] }
```

Figure 18: Level2 Methods to Hot-Swap Certain Modules

the outer scope until the inner-nested thread has terminated. Otherwise, `joinAndSpawn()` does not wait to join the subordinate thread and instead throws `IllegalArgumentException`.

Figure 18 provides the code for three methods that independently replace certain running modules within the level-3 component. All of the arguments to these three methods are declared with the `@Scoped` annotation. This means these arguments may refer to objects residing within temporary memory scopes. Because static analysis may not be able to determine all of the dynamic contexts within which these methods might be invoked, a run-time check is required on assignment to the `replace_class`, `replace_arg`, and `replace_parms` instance variables. This is why each of the methods has the `@AllowCheckedScopedLinks`

annotation. If the author of these components preferred to avoid this run-time check, he could remove the `@AllowCheckedScopedLinks` annotation and the `@Scoped` annotations on each of the arguments. This would force users of these services to pass references to `ImmortalMemory` objects as arguments.

## 8. Standard Hard Real-Time Extended API for Safety-Critical Java

Below, we describe each of the classes in the `javafx.realtime.util.sc` (safety-critical) package. Since these classes are not described elsewhere, each class includes detailed commentary to describe the behavior of its supported methods.

---



---

### AbsoluteTime.java

---

```
package javafx.realtime.util.sc;

/**
 *
 * RTSJ refinements:
 * ♦ public RelativeTime relative(); // removes clock argument
 */
public class AbsoluteTime extends javafx.realtime.AbsoluteTime {
    /**
     * Make a new AbsoluteTime object that has the same clock association and the same time
     * representation as the time parameter.
     *
     * Parameters:
     *
     * time: The AbsoluteTime object which is the source for the copy.
     *
     * throws java.lang.IllegalArgumentException if the time parameter is null.
     */
    public @StaticAnalyzable @ScopedPure AbsoluteTime(AbsoluteTime time)
        throws IllegalArgumentException;

    /**
     * Make a new AbsoluteTime object from the given AbsoluteTime object. The clock
     * association is made with the clock parameter. If clock is null, the association is made
     * with Clock.getRealtimeClock(). If the new copy has a different clock association than
     * this, any internal representation conversions are done only once, at the time the
     * copy is created.
     *
     * In contexts for which the constructed AbsoluteTime object is to be stack allocated, it
     * is the programmer's responsibility to assure that clock's lifetime is at least as long as
     * this newly constructed object. Since clock is not declared to be Scoped, it must reside
     * in ImmortalMemory within the safety-critical profile so this requirement is automatically
     * satisfied.
     *
     * Parameters:
     *
     * time: The AbsoluteTime object which is the source for the copy.
     *
     * clock: The clock providing the associaton for the newly constructed object.
     */
}
```

```
*
* throws java.lang.IllegalArgumentException if the time parameter is null.
*/
public @StaticAnalyzable @ScopedPure AbsoluteTime(AbsoluteTime time, Clock clock)
    throws IllegalArgumentException;

/**
 * Construct an AbsoluteTime object with time millisecond and nanosecond components
 * past the Realtime clock's Epoch (00:00:00 GMT on January 1, 1970) based on the parameter
 * millis plus the parameter nanos. The construction is subject to normalization of millis and
 * nanos parameters. If there is an overflow in the millisecond component when normalizing
 * then an IllegalArgumentException will be thrown. If after normalization the time object is
 * negative then the time represented by this is time before the Epoch.
 *
 * The clock association is implicitly made with Clock.getRealtimeClock().
 *
 * Parameters:
 *
 * millis: The desired value for the millisecond component of this. The actual value is the
 * result of parameter normalization.
 *
 * nanos: The desired value for the nanosecond component of this. The actual value is the
 * result of parameter normalization.
 *
 * throws java.lang.IllegalArgumentException if there is an overflow in the millisecond
 * component when normalizing.
 */
public @StaticAnalyzable @ScopedThis AbsoluteTime(long millis, int nanos)
    throws IllegalArgumentException;

/**
 * Construct an AbsoluteTime object with time millisecond and nanosecond components past
 * the epoch for clock. The value of the AbsoluteTime instance is based on the parameter
 * millis plus the parameter nanos. The construction is subject to normalization of millis and
 * nanos parameters. If there is an overflow in the millisecond component when normalizing
 * then an IllegalArgumentException will be thrown. If after normalization the time object is
 * negative then the time represented by this is time before the Epoch.
 *
 * The clock association is made with the clock parameter. If clock is null the association is
 * made with Clock.getRealtimeClock().
 *
 * In contexts for which the constructed AbsoluteTime object is to be stack allocated, it is
 * the programmer's responsibility to assure that clock's lifetime is at least as long as
 * this newly constructed object. Since clock is not declared to be Scoped, it must reside in
 * ImmortalMemory within the safety-critical profile so this requirement is automatically
 * satisfied.
 *
 * Parameters:
 *
 * millis: The desired value for the millisecond component of this. The actual value is the
 * result of parameter normalization.
 *
 * nanos: The desired value for the nanosecond component of this. The actual value is the
```

```

*      result of parameter normalization.
*
* clock: The clock providing the association for the newly constructed object.
*
* throws java.lang.IllegalArgumentException if there is an overflow in the millisecond
*      component when normalizing.
*/
public @StaticAnalyzable @ScopedThis AbsoluteTime(long millis, int nanos, Clock clock)
    throws IllegalArgumentException;

/**
 * This method is defined differently than in RTSJ. In the RTSJ, this method takes a
 * clock argument and creates a RelativeTime associated with that clock argument which
 * represents the difference between this and the current time. Though not detailed in
 * the RTSJ description of this method, in order to calculate the difference, it is necessary
 * to first convert "this" into a time representation that is consistent with the supplied
 * clock argument. The behavior of this method within the RTSJ is inconsistent with other
 * AbsoluteTime difference methods, which are generally defined to throw an
 * IllegalArgumentException} if the clocks associated with the two objects to be compared
 * are different.
 *
 * So the suggested API here is to remove the clock argument and require that the returned
 * RelativeTime result is associated with the same clock as this AbsoluteTime object.
 *
 * Convert the time of this to a relative time by subtracting the current time from this time.
 * A destination object is allocated to return the result. The clock associated with the result
 * is the same clock that is associated with this object.
 *
 * In contexts for which the returned RelativeTime object is to be stack allocated, it is the
 * programmer's responsibility to assure that the associated clock's lifetime is at least as long
 * as this newly constructed object. Since there are no APIs that would allow creation of an
 * AbsoluteTime object with a reference to a Scoped clock, we are assured within the safety-
 * critical profile that the clock resides in ImmortalMemory, so this requirement is automati-
 * cally satisfied.
 *
 * returns the RelativeTime conversion results in a newly allocated object associated with
 *      the same clock as this.
*/
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis RelativeTime relative();

/**
 * This method is different than the RTSJ in that it does not expect a Clock argument. In
 * general, we would expect any automatic conversion from an existing HighResolutionTime
 * to an AbsoluteTime would need to be associated with the same Clock. Assuming that we
 * want to enforce this restriction, the Clock argument is redundant.
 *
 * Convert the time of this to an absolute time. The returned AbsoluteTime object will be
 * associated with the same Clock as this. See the derived class comments for more specific
 * information.
 *
 * returns the AbsoluteTime conversion in a newly allocated object.
*/
public @StaticAnalyzable @ScopedThis @CallerAllocatedResult AbsoluteTime absolute();

```

```
}
```

---

---

## AperiodicParameters.java

---

```
package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

public class AperiodicParameters extends javax.realtime.AperiodicParameters {
    /**
     * Construct a new AperiodicParameters object with the specified deadline and miss_handler.
     * The timing accuracy for detection of deadline overrun shall be the first tick of the
     * default real-time clock that follows the desired deadline. In other words, triggering of the
     * miss_handler may be delayed by up to the amount time represented by the value returned
     * from Clock.getResolution() for the default real-time clock, which is represented by
     * Clock.getRealtimeClock(). The queue overflow behavior is set to the default value of
     * arrivalTimeQueueOverflowIgnore and the queue length is set to one.
     *
     * Parameters:
     *
     * deadline: The latest permissible completion time measured from the release time of the
     * associated invocation of the schedulable object. If null, the default value of
     * RelativeTime.UNDEFINED_TIME is used. This denotes that no deadline should be
     * enforced.
     *
     * miss_handler: The handler that is invoked if the run() method of the schedulable object is
     * still executing after the deadline has passed. If null, the default value is no miss
     * handler.
     */
    public @StaticAnalyzable @ScopedPure
        AperiodicParameters(RelativeTime deadline, AsyncEventHandler miss_handler);

    /**
     * Construct a new AperiodicParameters object with the specified deadline, miss_handler,
     * queue_length, and overflow_behavior.
     *
     * Parameters:
     *
     * deadline: The latest permissible completion time measured from the release time of the
     * associated invocation of the Schedulable object. If null, the default value of
     * RelativeTime.UNDEFINED_TIME is used. This denotes that no deadline should
     * be enforced.
     *
     * Enforcement of deadline compliance is limited by the granularity of the default
     * real-time clock's tick period. At one extreme, an event is fired immediately before
     * a timer tick. This means the first tick following the event firing must be assumed to
     * represent the passage of zero time. At the other extreme, the corresponding event
     * is fired immediately following a tick. Now, because we don't count the first tick as
     * passage of any time, in this case, deadline enforcement will underestimate the
     * passage of time since event firing by up to the amount of time in the default real-
     * time clock's tick interval. In other words, the system will understand that after the
     * second tick has been processed, only one tick interval has passed, even though two
```

```

*      tick intervals have passed in reality. Now, assume that the event handler finishes
*      immediately before another timer tick. This means the measurement of time
*      required to fully handle the asynchronous event is underestimated by essentially
*      another full timer tick. In summary, because of limits in the system's ability to
*      accurately measure time, an asynchronous event may miss its deadline by up to twice
*      the default clock's tick interval minus an arbitrarily small delta without triggering
*      execution of the deadline miss handler.
*
* miss_handler: The handler that is invoked if the run() method of the schedulable object is
*      still executing after the deadline has passed. If null, the default value is no miss
*      handler.
*
* queue_length: Each AsyncEventHandler keeps track of the time at which each pending
*      triggered event was fired. It uses this information to detect deadline overruns.
*      This parameter specifies the length of this queue. The queue must have at least one
*      entry.
*
* overflow_behavior: When the arrival-time queue overflows, the behavior can be configured
*      to either ignore the new event, or to replace the oldest pending event with the
*      new event. These two responses are selected by passing as this argument either the
*      value of arrivalTimeQueueOverflowIgnore or arrivalTimeQueueOverflowReplace
*      respectively.
*
* throws java.lang.IllegalArgumentException if deadline is less than or equal to zero, or if the
*      queue_length argument is less than one, or if the value of overflow_behavior does
*      not equal one of arrivalTimeQueueOverflowIgnore or arrivalTimeQueueOverflow-
*      Replace.
*/
public @StaticAnalyzable @ScopedPure AperiodicParameters(RelativeTime deadline,
    AsyncEventHandler miss_handler, int queue_length, String overflow_behavior);

/**
 * Gets a reference to the arrival_time_queue_overflow_behavior configuration.
 *
 * returns a reference to the value of arrival_time_queue_overflow_behavior.
 */
public @StaticAnalyzable @ScopedThis String getArrivalTimeQueueOverflowBehavior();

/**
 * Gets the initial (and final, because this state variable cannot change) number of entries that
 * the arrival time queue can hold.
 *
 * returns the length of the arrival-time queue.
 */
public @StaticAnalyzable @ScopedThis int getInitialArrivalTimeQueueLength();
}

```

---

## AsyncEvent.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

```

```

import javax.realtime.UnknownHappeningException;

/**
 * This class mimics the behavior of javax.realtime.AsyncEvent. However, it differs in that the
 * type is immutable. This means we have to provide all configuration information at the time an
 * object is instantiated.
 *
 *
 * RTSJ Additions:
 * ♦ public AsyncEvent(AsyncEventHandler handler);
 * ♦ public AsyncEvent(AsyncEventHandler handlers[]);
 * ♦ public AsyncEvent(String happening, AsyncEventHandler handler);
 * ♦ public AsyncEvent(String happening, AsyncEventHandler handlers[]);
 */
public class AsyncEvent extends javax.realtime.AsyncEvent {

    /**
     * Create a new AsyncEvent object to trigger the corresponding handler each time it is fired.
     *
     * Parameters:
     *
     * handler: the event handler that is triggered each time this event is fired.
     *
     * throws IllegalArgumentException if the handler is null.
     */
    public @StaticAnalyzable @ScopedPure AsyncEvent(AsyncEventHandler handler)
        throws IllegalArgumentException;

    /**
     * Create a new AsyncEvent object to trigger each of the event handlers in the handlers
     * array each time this event is fired. A private copy of the handlers array is allocated within
     * the current allocation context.
     *
     * Parameters:
     *
     * handlers: an array of event handlers, all of which are to be triggered for execution each
     * time this event is fired.
     *
     * throws IllegalArgumentException if handlers is null or if any entry in handlers is null, or if
     * any entry in handlers appears more than once.
     */
    public @ScopedPure AsyncEvent(AsyncEventHandler handlers[])
        throws IllegalArgumentException;

    /**
     * Create a new AsyncEvent object to be bound to an implementation-defined happening that
     * occurs outside the Java environment (e.g. an operating system power-down signal). Each
     * time this happening occurs, we trigger the corresponding handler to be executed.
     *
     * Parameters:
     *
     * happening: An implementation-dependent value representing some external event to which
     * this instance of AsyncEvent is bound.
     *
     *
     */

```

```

* handler: The event handler that is triggered each time this event is fired.
*
* throws IllegalArgumentException if the handler is null.
*
* throws UnknownHappeningException if the specified happening is not understood for this
*     execution environment.
*/
public @StaticAnalyzable @ScopedPure
    AsyncEvent(String happening, AsyncEventHandler handler)
        throws IllegalArgumentException, UnknownHappeningException;

/**
* Create a new AsyncEvent object to be bound to an implementation-defined happening that
* occurs outside the Java environment (e.g. an operating system power-down signal). Each
* time this happening occurs, we trigger each of the event handlers in the handlers array to
* be executed.
*
* Parameters:
*
* happening: An implementation-dependent value representing some external event to which
*     this instance of AsyncEvent is bound.
*
* handlers: An array of event handlers, all of which are to be triggered for execution each
*     time this event is fired.
*
* throws IllegalArgumentException if handlers is null or if any entry in handlers is null, or if
*     any entry in handlers appears more than once.
*
* throws UnknownHappeningException if the specified happening is not understood for this
*     execution environment.
*/
public @ScopedPure AsyncEvent(String happening, AsyncEventHandler handlers[])
    throws IllegalArgumentException;
}

```

---

## Atomic.java

---

```

package javax.realtime.util.sc;

/**
* Atomic denotes the same special handling that is associated with PCP, with the following
* enhancements:
*
* 1. The body of any synchronized method (or statement if synchronized statements are
*     allowed in a mission-critical superset of the safety-critical profile) must be execution-
*     time bounded (StaticAnalyzable in all modes of analysis).
*
* 2. Code within synchronized methods is interruption deferred.
*/
public interface Atomic extends PCP {
}

```

---

---

**BoundAsyncEventHandler.java**

---

```
package javax.realtime.util.sc;

import javax.realtime.SchedulingParameters;
import javax.realtime.ReleaseParameters;
import javax.realtime.MemoryParameters;
import javax.realtime.NoHeapRealtimeThread;

/**
 * This class mimics javax.realtime.BoundAsyncEventHandler.
 * <p>
 * RTSJ additions:
 * ♦ BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
 *   MemoryParameters memory, MemoryArea area, boolean is_daemon);
 * ♦ NoHeapRealtimeThread proxy();
 */
public class BoundAsyncEventHandler extends javax.realtime.BoundAsyncEventHandler {
    /**
     * Create an instance of BoundAsyncEventHandler with the specified parameters.
     * Note that we do not take a MemoryArea argument because the safety-critical profile only
     * has one memory area, which is ImmortalMemory.
     *
     * When a BoundAsyncEventHandler is created, a corresponding NoHeapRealtimeThread is
     * also created and the two are bound together. This NoHeapRealtimeThread is known as a
     * proxy thread. However, it is not considered part of the general-purpose proxy pool, and it
     * does not contribute to the counts returned by AsyncEventHandler.numProxyServers(int),
     * AsyncEventHandler.numActiveProxies(int), and AsyncEventHandler.numBlocked-
     * Proxies(int).
     *
     * Parameters:
     *
     * scheduling: A SchedulingParameters object which will be associated with the constructed
     * instance. This must be non-null. Note that the only scheduling supported by the
     * safety-critical profile is fixed-priority without time slicing.
     *
     * release: A ReleaseParameters object which will be associated with the constructed
     * instance. This must be non-null. Note that the only release parameters supported by
     * the safety-critical profile are for periodic, aperiodic and sporadic threads. The
     * difference between aperiodic and sporadic is that sporadic specifies a minimum
     * inter-arrival time for consecutive firings.
     *
     * memory: A MemoryParameters object which will be associated with the constructed
     * instance. This must be non-null.
     *
     * is_daemon: If true, marks this event handler as a daemon event handler. If false, marks
     * this event handler or a user event handler. The Real-Time Virtual Machine exits
     * when the only schedulable objects and threads running are all daemon.
     *
     * throws IllegalArgumentException if priority set to interrupt level or is otherwise outside
     * the range of legal priority levels.
     *
     */
}
```

(Permission granted to reproduce and distribute as a complete document, without modification)

```
* throws NullPointerException if any of its arguments equal null.
*/
public @StaticAnalyzable @ScopedPure
    BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
        MemoryParameters memory, boolean is_daemon)
        throws IllegalArgumentException;

/**
 * Returns the proxy thread that is bound to this BoundAsyncEventHandler object.
 *
 * returns the proxy thread.
 */
public @StaticAnalyzable NoHeapRealtimeThread proxy();
}
```

---

---

### CallerAllocatedArrayResult.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies methods that are declared to return an object reference, for which
 * the referenced object may be allocated in the context of the caller's run-time stack. This
 * annotation imposes certain restrictions on the implementation of the method.
 *
 * In all cases, the caller must set aside the memory to hold the referenced object. In the case
 * that the caller does not desire to follow the restrictive practice required for stack allocation
 * of the object, it shall set aside heap memory rather than stack memory to hold the method's
 * result.
 */
@Documented @Target({ElementType.METHOD}) @Retention(RetentionPolicy.CLASS)
@Inherited public @interface CallerAllocatedArrayResult {
    java.lang.Class [] subclasses() default {};
}
```

---

---

### CallerAllocatedResult.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies methods that are declared to return an object reference, for which
 * the referenced object may be allocated in the context of the caller's run-time stack. This
 * annotation imposes certain restrictions on the implementation of the method.
 *
 * In all cases, the caller must set aside the memory to hold the referenced object. In the case
 * that the caller does not desire to follow the restrictive practice required for stack allocation
 * of the object, it shall set aside heap memory rather than stack memory to hold the method's
 * result.
 */
```

```

*/
@Documented @Target({ElementType.METHOD}) @Retention(RetentionPolicy.CLASS)
@Inherited public @interface CallerAllocatedResult {
    java.lang.Class [] subclasses() default {};
}

```

## Ceiling.java

```

package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * Annotate a class's ceilingPriority() method to identify the expected ceiling priority to be
 * associated with instances of this class. The byte-code verifier rejects uses of this annotation
 * for any class that does not implement javax.realtime.util.sc.PCP and any method that does not
 * match the name and signature of:
 *
 *     public int ceilingPriority();
 *
 * In the safety-critical profile, it also rejects programs for which the value of the ceiling
 * attribute is less than 1 or greater than 28 (or 128 for mission-critical compatibility).
 */
@Target({ElementType.METHOD}) @Retention(RetentionPolicy.RUNTIME) @Documented
@Inherited public @interface Ceiling {
    /**
     * Specifies the ceiling priority that we expect to associate with every instance of the enclosing
     * class. At instantiation time, the constructor will terminate by throwing an IllegalState-
     * Exception if the current default monitor control policy does not equal PriorityCeiling-
     * Emulation or its corresponding ceiling priority does not equal the value of this attribute, or
     * if the value returned from the ceilingPriority() method does not equal the value of this
     * attribute.
     *
     * Note that a typical safety-critical application instantiates all of its lock objects at link time
     * rather than run time. Thus, a programming error in which programmers attempt to
     * instantiate a locking object at the incorrect ceiling priority will normally be reported as a
     * link-time error rather than a run-time error. We would expect to have a lint-like analysis
     * tool available for the purpose of identifying to programmers circumstances in which a
     * locking object that is created at run-time might possibly result in a run-time error because
     * of inconsistencies in the object's ceiling priority specification.
     */
    int value() default 28;
}

```

## Clock.java

```

package javax.realtime.util.sc;

import javax.realtime.AbsoluteTime;
import javax.realtime.RelativeTime;

```

```

/**
 * This class mimics javax.realtime.Clock. However, there are a number of noteworthy
 * differences. First, key philosophical differences are enumerated here:
 *
 * ♦ We consider it very important to assure some level of consistency in Clock services
 * across all compliant implementations of the safety-critical Java subset. Thus, we require
 * that all compliant implementations have the ability to represent (or at least emulate) the
 * epoch according to the conventions of the local time representation.
 *
 * ♦ We recognize that it is very common for embedded hardware to offer two very
 * different implementations of a clock. With one broad class of clock implementations, a
 * fixed interval countdown timer generates a hardware interrupt once per period. Besides
 * using this hardware interrupt to increment the computer's internal representation of
 * time, the interrupt handler can also take responsibility for triggering the execution of
 * certain code, such as is necessary when we overrun a deadline, or release a sleeping
 * thread, or trigger execution of a periodic task. The other broad class of timing services
 * is a hardware clock that keeps track of the passage of time in private hardware
 * registers, but does not trigger execution of time-driven code. This hardware clock must
 * be polled to determine the current time. We require that every compliant implementa-
 * tion of safety-critical Java provide two default clocks, one representing each of these
 * two broad classes of timers. This enables more precise time accounting on systems
 * that offer hardware clocks. On systems that do not have hardware clocks, the hardware
 * clock can be emulated using timer ticks from the fixed interval countdown timer (which
 * is assumed to exist in all systems). In the case that a fixed interval countdown timer is
 * used to emulate a hardware clock, this same Clock object can be returned both from
 * getRealtimeClock() and getTimeKeeper().
 *
 * ♦ Some hardware platforms support efficient variable interval countdown timers. For
 * systems that offer this capability, the count-down timer's resolution is listed to be the
 * minimum time between consecutive firings. Each time the tick handler completes
 * execution, it sets the count-down timer to skip intervening ticks, generating the next
 * tick to occur only when the next time-triggered work (sleeping threads need to be
 * awakened, deadlines need to be enforced, periodic threads need to be triggered) is
 * ready to be performed. This represents an optimization of the active clock, in that we
 * avoid consuming CPU time and memory cache with each timer tick.
 *
 * RTSJ additions:
 *
 * ♦ Clock(RelativeTime resolution);
 * ♦ Clock(RelativeTime resolution, boolean is_active);
 * ♦ isActive();
 * ♦ getTimeKeeper();
 */
public class Clock extends javax.realtime.Clock {

    /**
     * Parameters:
     *
     * ♦ resolution: Desired resolution of this clock.
     *
     * ♦ throws IllegalArgumentException if the requested resolution is not available.
     */
    protected @StaticAnalyzable Clock(RelativeTime resolution, boolean is_active)

```

```
    throws IllegalArgumentException;

/**
 * Determine whether this Clock object is active, meaning it is capable of driving time-driven
 * activities by associating this Clock with a Timer object.
 *
 * returns true if this Clock can be associated with Timer objects to drive time-driven
 *          execution of software.
 */
public @StaticAnalyzable boolean isActive();

/**
 * There is always at least one passive clock object available: a real-time clock that advances
 * in sync with the external world. This is the default passive Clock. This default clock is
 * monotonically increasing. It is impervious to user actions that might "correct" the host
 * computer system's wall-clock time, or to corrections in the local time because of daylight
 * savings adjustments, or because of global clock synchronization algorithms. Note that a
 * passive clock is a clock for which it is possible to request the current time, but for which it
 * may not be possible to trigger execution of particular activities at specific future times
 * associated with this Clock.
 *
 * returns an instance of the default Clock. This default real-time clock always resides in
 * ImmortalMemory.
 */
public @StaticAnalyzable static Clock getTimeKeeper();
}
```

---

## Configuration.java

---

```
package javax.realtime.util.sc;

/**
 * Each implementation, and each platform, may exhibit different default values for all of the
 * various configuration parameters.
 *
 * The main program must instantiate the primordial Configuration object within its static
 * initializer code. All other initialization is dependent upon successful initialization of the
 * Configuration object.
 */
public class Configuration {

    /**
     * Is this a multi-processor environment?
     */
    public static final boolean MULTIPROCESSOR;

    /**
     * How many SMP processors in this run-time environment?
     */
    public static final int NUM_PROCESSORS;

    /**
```

```
* The number of nanoseconds in a clock tick for the default real-time clock.
*/
public final RelativeTime TICK_DURATION;

/**
 * The default number of bytes of memory for a new NoHeapRealtimeThread's stack.
 */
public final int DEFAULT_STACK_SIZE;

/**
 * If true, every method invocation includes code to assure that the run-time stack is large
 * enough to reliably support execution of the invoked method. Otherwise, no stack checking
 * is performed. In the latter case, stack overflow will have unpredictable results.
 */
public final boolean STACK_OVERFLOW_CHECKING;

/**
 * If true, when stack overflow is detected, the stack of the current thread will be expanded,
 * assuming that the given thread's stack limit has not been exceeded. If false, the stack will
 * never grow beyond its initial size.
 */
public final boolean STACK_EXPANSION_ENABLED;

/**
 * If STACK_EXPANSION_ENABLED is true, this constant represents the number of bytes
 * added to the stack each time the run-time stack is expanded. Note that expansion of the
 * stack need not be contiguous. The new stack segment may be allocated in a different region
 * of memory, with a trampoline function serving the purpose of reclaiming this stack
 * segment's memory and adjusting the stack and frame pointers to refer to the original
 * stack segment when control returns to (or through, by way of a thrown exception) the
 * trampoline function.
 */
public final int STACK_EXPANSION_INCREMENT;

/**
 * If true, any array subscript operations within methods that are accompanied by the
 * @OmitSubscriptChecking annotation will not perform subscript checks. If false, the array
 * subscript operations will still be checked in spite of the programmer's annotation.
 *
 * Array subscript checking within methods that do not include the OmitSubscriptChecking
 * annotation are unaffected by this configuration variable. Those subscript operations are
 * always checked.
 */
public final boolean ELIDE_ARRAY_SUBSCRIPT_CHECKING;

/**
 * If true, any array subscript operations within methods that are accompanied
 * by the @OmitScopeChecking annotation will not perform scope assignment
 * checks. If false, the assignment operations will still be checked in
 * spite of the programmer's annotation.
 * <p>
 * Assignment scope checking within methods that do not include the
 * OmitScopeChecking annotation are unaffected by this configuration
```

```
* variable. Those assignment operations are always checked.
*/
public static final boolean ELIDE_SCOPE_CHECKING = false;

/**
 * The number of bytes of memory dedicated to the global ImmortalMemory pool. Any objects
 * allocated during static initialization by the linker will be placed in static memory which is
 * never reclaimed, but which is not considered to be part of the ImmortalMemory heap.
 * Objects allocated during the class initialization that occurs during startup (after static
 * linking) are allocated within the ImmortalMemory heap.
 */
public final long IMMORTAL_SIZE;

/**
 * We maintain a private array of N integers, each entry of which represents the operating
 * system priority to which the corresponding safety-critical Java thread priority maps. For
 * example, if priority_map[0] equals 16, this means safety-critical Java thread priority 1 maps
 * to operating system priority 16. N equals 28 for the safety-critical configuration, and 128
 * for the mission-critical configuration.
 *
 * We keep this information private because we do not want application code to modify the
 * values stored in this array. Application code may request copies of the array contents by
 * invoking getSystemPriorityMap().
 */
private final int[] PRIORITY_MAP;

/**
 * The number of priorities that are dedicated to interrupt handling. This number of priorities
 * are reserved at the top end of the range of available priorities. No java.lang.Thread or
 * AsyncEventHandler is allowed to set its priority to these levels. Only interrupt handlers
 * and priority ceiling emulation locks are allowed to specify these priority levels.
 *
 * This value is a characteristic of the hardware and its integration with the safety-critical
 * Java executive. It is not user configurable.
 */
public final int NUM_INTERRUPT_PRIORITIES;

/**
 * If true, this computer uses little-endian information representation. Otherwise, it uses big-
 * endian representations.
 */
public final boolean LITTLE_ENDIAN;

/**
 * The domain name of the vendor who supplied this particular safety-critical Java run-time
 * environment, as in "com.aonix".
 */
public final String VENDOR_NAME;

/**
 * The product version description, using vendor-specific conventions, as in "1.0;Build 387"
 */
public final String PRODUCT_VERSION;
```

```
/**
 * The specification version to which this safety-critical Java environment conforms, as in
 * "0.9"
 */
public final String SPECIFICATION_VERSION;

/**
 * This returns a copy of the private PRIORITY_MAP array. This array holds M entries to
 * represent each of the priorities supported by the safety-critical Java profile. Each entry
 * represents the underlying real-time operating system priority to which the Java priority is
 * mapped. For example, PRIORITY_MAP[0] represents the operating system priority
 * at which Java priority 1 threads run. In the safety-critical configuration, M is 28. In the
 * mission-critical configuration, M is 128.
 *
 * If this safety-critical Java profile is implemented on bare hardware, without an underlying
 * real-time operating system kernel, then each entry maps to the same value as the Java
 * priority. In other words, PRIORITY_MAP[N]=N+1.
 *
 * For any priorities that map to interrupt handlers, the corresponding entry within the
 * PRIORITY_MAP equals zero.
 *
 * returns anewly allocated array copy of the private PRIORITY_MAP.
 */
public final @StaticAnalyzable @CallerAllocatedResult int[] getSystemPriorityMap();

/**
 * This returns the system-level priority number that corresponds to the supplied java priority
 * represented by the java_priority_no argument.
 *
 * Parameters:
 *
 * java_priority_no: The Java priority number for which we desire to know the corresponding
 * RTOS priority number.
 *
 * returns the RTOS priority number that corresponds to Java priority java_priority_no.
 *
 * throws IllegalArgumentException if java_priority_no < 1 or java_priority_no > M.
 */
public final @StaticAnalyzable int getSystemPriority(int java_priority_no)
    throws IllegalArgumentException;

/**
 * Construct this system's primordial configuration, using default values for arguments. This is
 * private, because it's only to be used by this class's static factory methods.
 *
 * Parameters:
 *
 * immortal_size: The desired size, in bytes, of the ImmortalMemory region.
 *
 * array_subscript_checking: If true, the generated code performs array subscript checking.
 * If false, the generated code does not check for array subscript violations.
 */
```

```

*
* tick_duration: The desired tick period for the default real-time clock.
*
* default_stack_size: The number of bytes in a newly allocated run-time stack, by default.
*
* stack_overflow_checking: If true, the safety-critical Java environment is configured to
*   perform stack overflow checking with each method invocation. If false, the
*   safety-critical Java environment performs no stack overflow checking. Stack over-
*   flow checking is typically performed by method prologue code that is generated by
*   the AOT compiler.
*
* stack_expansion_enabled: If true, a thread that overflows its run-time stack will attempt
*   to expand its stack. If false, or if the stack expansion is unsuccessful, a thread that
*   overflows its stack will throw StackOverflowError.
*
* stack_expansion_increment: If stack_expansion_enabled is true, this parameter
*   represents the number of bytes by which to expand the stack each time it expands.
*
* elide_array_subscript_checking: If true, the safety-critical Java environment is
*   configured to ignore array subscript checking within methods that are declared with
*   the @OmitSubscriptChecking annotation.
*
* priorities: For each of the M Java priorities, the corresponding entry of this array
*   represents the real-time operating system priority at which threads of this Java
*   priority run. For example, if priorities[3] holds the value 10, this means that safety-
*   critical Java threads with safety-critical thread priority 4 run at real-time operating
*   system priority 10.
*
* throws IllegalStateException if the singleton Configuration object has already been
*   instantiated
*
* throws IllegalArgumentException if the arguments are incompatible with the execution
*   platform
*/
private @StaticAnalyzable Configuration(long immortal_size,
    boolean array_subscript_checking, RelativeTime tick_duration, int default_stack_size,
    boolean stack_overflow_checking, boolean stack_expansion_enabled,
    int stack_expansion_increment, boolean elide_array_subscript_checking,
    int priorities[]) throws IllegalStateException, IllegalArgumentException;

/**
* Construct this system's primordial configuration, using the supplied parameters to initialize
* the configuration.
*
* Parameters:
*
* tick_duration: The desired tick period for the default real-time clock.
*
* throws IllegalStateException if the singleton Configuration object has already been
*   instantiated
*
* throws IllegalArgumentException if the argument is incompatible with the execution
*   platform

```

```
*/
public static @StaticAnalyzable
    Configuration make_configuration(RelativeTime tick_duration)
        throws IllegalStateException, IllegalArgumentException;

/**
 * Construct this system's primordial configuration, using the supplied parameters to initialize
 * the configuration.
 *
 * Parameters:
 *
 * tick_duration: The desired tick period for the default real-time clock.
 *
 * stack_overflow_checking: If true, the safety-critical Java environment is configured to
 * perform stack overflow checking with each method invocation. If false, the
 * safety-critical Java environment performs no stack overflow checking. Stack
 * overflow checking is typically performed by method prologue code that is generated
 * by the AOT compiler.
 *
 * throws IllegalStateException if the singleton Configuration object has already been
 * instantiated
 *
 * throws IllegalArgumentException if the arguments are incompatible with the execution
 * platform
 */
public static @StaticAnalyzable
    Configuration make_configuration(RelativeTime tick_duration,
        boolean stack_overflow_checking)
        throws IllegalStateException, IllegalArgumentException;

/**
 * Construct this system's primordial configuration, using the supplied parameters to initialize
 * the configuration.
 *
 * Parameters:
 *
 * immortal_size The desired size, in bytes, of the ImmortalMemory region.
 *
 * Parameters:
 *
 * array_subscript_checking: If true, the generated code performs array subscript checking.
 * If false, the generated code does not check for array subscript violations.
 *
 * tick_duration: The desired tick period for the default real-time clock.
 *
 * default_stack_size: The number of bytes in a newly allocated run-time stack, by default.
 *
 * stack_overflow_checking: If true, the safety-critical Java environment is configured to
 * perform stack overflow checking with each method invocation. If false, the
 * safety-critical Java environment performs no stack overflow checking. Stack over-
 * flow checking is typically performed by method prologue code that is generated by
 * the AOT compiler.
 */
```

```

* stack_expansion_enabled: If true, a thread that overflows its run-time stack will attempt
*   to expand its stack. If false, or if the stack expansion is unsuccessful, a thread that
*   overflows its stack will throw StackOverflowError.
*
* stack_expansion_increment: If stack_expansion_enabled is true, this parameter
*   represents the number of bytes by which to expand the stack each time it expands.
*
* priorities: For each of the 28 safety-critical Java priorities, the corresponding entry of
*   this array represents the real-time operating system priority at which threads of
*   this Java priority run. For example, if priorities[3] holds the value 10, this means
*   that safety-critical Java threads with safety-critical thread priority 4 run at real-
*   time operating system priority 10.
*
* throws IllegalStateException if the singleton Configuration object has already been
*   instantiated
*
* throws IllegalArgumentException if the arguments are incompatible with the execution
*   platform
*/
public static @StaticAnalyzable
    Configuration make_configuration(long immortal_size,
        boolean array_subscript_checking,
        RelativeTime tick_duration,
        int default_stack_size,
        boolean stack_overflow_checking,
        boolean stack_expansion_enabled,
        int stack_expansion_increment,
        int priorities[])
    throws IllegalStateException, IllegalArgumentException;

/**
* returns the singleton Configuration object
*/
public static final @StaticAnalyzable Configuration instance();
}

```

---

## EmbeddedConflictException.java

---

```

package javax.realtime.util.sc;

/**
* Thrown to indicate that a request to perform some embedded operations (such as opening of a
* low-level I/O port) cannot be satisfied because it violates a system invariant.
*/
public class EmbeddedConflictException extends java.lang.Exception {

    /**
    * Construct a new EmbeddedConflictException object.
    */
    public @StaticAnalyzable @ScopedThis EmbeddedConflictException();

    /**

```

```
* Construct a new EmbeddedConflictException with the specified detail message.
*
* Parameters:
*
* msg the detail message.
*/
public @StaticAnalyzable @ScopedPure EmbeddedConflictException(String msg);
}
```

---

## FinalSizeEstimator.java

---

```
package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

/**
 * This is like SizeEstimator, except the object is considered immutable. Once instantiated, its
 * contents cannot be changed. All of the methods that would normally mutate its contents will
 * instead throw UnsupportedOperationException.
 *
 * see javax.realtime.util.sc.SizeEstimator
 */
public class FinalSizeEstimator extends SizeEstimator {

    /**
     * Construct a new FinalSizeEstimator object, with initial values copied from the constructor's
     * value argument.
     *
     * Parameters:
     *
     * value provides the initial (and final constant) values of this FinalSizeEstimator object.
     */
    public @StaticAnalyzable @ScopedThis FinalSizeEstimator(SizeEstimator value);

    /**
     * Represents the estimated number of bytes of memory
     *
     * returns The estimated size in bytes.
     */
    public @StaticAnalyzable @ScopedThis long getEstimate();

    /**
     * This method ignores its arguments and always throws a previously allocated instance of
     * UnsupportedOperationException because all internal state information is considered to
     * be immutable.
     *
     * Parameters:
     *
     * c: The class describing the type of each array element.
     */
}
```

```

*
* number: The number of elements in the array.
*/
public @StaticAnalyzable @ScopedThis void reserve(@Scoped java.lang.Class c, int number)
    throws UnsupportedOperationException;

/**
* This method ignores its arguments and always throws previously allocated instance of
* UnsupportedOperationException because all internal state information is considered to be
* immutable.
*
* Parameters:
*
* size: The given instance of SizeEstimator to be summed to the total memory size
*         represented by this object.
*/
public @StaticAnalyzable @ScopedThis void reserve(SizeEstimator size)
    throws UnsupportedOperationException;

/**
* This method ignores its arguments and always throws a previously allocated instance of
* UnsupportedOperationException because all internal state information is considered to
* be immutable.
*
* Parameters:
*
* estimator: The given instance of SizeEstimator.
*
* number: The number of times to reserve the size denoted by estimator.
*/
public @StaticAnalyzable @ScopedThis
    void reserve(@Scoped SizeEstimator estimator, int number)
    throws UnsupportedOperationException;

/**
* This method ignores its arguments and always throws a previously allocated instance of
* UnsupportedOperationException because all internal state information is considered to
* be immutable.
*
* Parameters:
*
* dimension: The dimension of the array.
*/
public @StaticAnalyzable @ScopedThis void reserveArray(int dimension)
    throws UnsupportedOperationException;

/**
* This method ignores its arguments and always throws a previously allocated instance of
* UnsupportedOperationException because all internal state information is considered to
* be immutable.
*
* Parameters:
*

```

```

* dimension: The dimension of the array.
*
* type: The class representing a primitive type. The reservation will leave room for an array
*       of dimension elements, with each element of this type.
*/
public @StaticAnalyzable @ScopedThis
    void reserveArray(int dimension, @Scoped java.lang.Class type)
        throws UnsupportedOperationException;

/**
 * This method ignores its arguments and always throws a previously allocated instance of
 * UnsupportedOperationException because all internal state information is considered to
 * be immutable.
 *
 * Parameters:
 *
 * number: The number of bytes to add in to the total.
 */
public @StaticAnalyzable @ScopedThis
    void reserveBytes(int number) throws
        IllegalArgumentException, UnsupportedOperationException;
/**
 * This method ignores its arguments and always throws a previously allocated instance of
 * UnsupportedOperationException because all internal state information is considered to
 * be immutable.
 *
 * Parameters:
 *
 * aeh: The AsyncEventHandler to be analyzed for determination of its stack memory
 *       requirements.
 *
 * throws IllegalArgumentException if the aeh implementation cannot be statically analyzed.
 */
public @ScopedThis void reserveStack(@Scoped AsyncEventHandler aeh)
    throws IllegalArgumentException, UnsupportedOperationException;

/**
 * This method ignores its arguments and always throws a previously allocated instance of
 * UnsupportedOperationException because all internal state information is considered to
 * be immutable.
 *
 * Parameters:
 *
 * runcode: The Runnable to be analyzed for determination of its stack memory requirements.
 *
 * throws IllegalArgumentException if the runcode implementation cannot be statically
 *       analyzed.
 */
public @ScopedThis void reserveStack(@Scoped Runnable runcode)
    throws IllegalArgumentException, UnsupportedOperationException;
}

```

---



---

## HighResolutionTime.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AbsoluteTime;
import javax.realtime.RelativeTime;

/**
 * This class mimics javax.realtime.HighResolutionTime. It differs in that instances of the class
 * are immutable.
 *
 * RTSJ refinements:
 *
 * ♦ RelativeTime relative(Clock clock); // removed clock argument
 * ♦ AbsoluteTime absolute(Clock clock); // removed clock argument
 */
public abstract class HighResolutionTime
    extends javax.realtime.HighResolutionTime implements java.lang.Comparable {

    /**
     * This method is different than the RTSJ in that it does not expect a Clock argument. In
     * general, we would expect any automatic conversion from an existing HighResolutionTime to
     * an AbsoluteTime would need to be associated with the same Clock. Assuming that we want
     * to enforce this restriction, the Clock argument is redundant.
     *
     * Convert the time of "this" to an absolute time. The returned AbsoluteTime object will be
     * associated with the same Clock as this. See the derived class comments for more
     * specific information.
     *
     * returns The AbsoluteTime conversion in a newly allocated object.
     */
    public abstract @StaticAnalyzable @ScopedThis @CallerAllocatedResult
        AbsoluteTime absolute();

    /**
     * This method is defined differently than in RTSJ. In the RTSJ, this method takes a clock
     * argument and creates a RelativeTime associated with that clock argument which represents
     * the difference between this and the current time. Though not detailed in the RTSJ
     * description of this method, in order to calculate the difference, it is necessary to first
     * convert this into a time representation that is consistent with the supplied clock argument.
     * The behavior of this method within the RTSJ is inconsistent with other AbsoluteTime
     * difference methods, which are generally defined to throw an IllegalArgumentException if
     * the clocks associated with the two objects to be compared are different.
     *
     * So the suggested API here is to remove the clock argument and require that the returned
     * RelativeTime result is associated with the same clock as this AbsoluteTime object.
     *
     * Convert the time of this to a relative time by subtracting the current time from this time.
     * A destination object is allocated to return the result. The clock associated with the result
     * is the same clock that is associated with this object.
     *
     * In contexts for which the returned RelativeTime object is to be stack allocated, it is the

```

```

* programmer's responsibility to assure that the associated clock's lifetime is at least as long
* as this newly constructed object. Since there are no APIs that would allow creation of an
* AbsoluteTime object with a reference to a @Scoped clock, we are assured within the
* safety-critical profile that the clock resides in ImmortalMemory, so this requirement is
* automatically satisfied.
*
* returns a newly allocated RelativeTime object associated with the same Clock as this.
*/
public abstract @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    RelativeTime relative();
}

```

---

## IOPort.java

---

```

package javax.realtime.util.sc;

/**
 * This class and its descendents have no analog in the RTSJ. PhysicalMemory in the RTSJ is
 * designed to serve different purposes.
 *
 * The purpose served by this class is to abstract access to a physical I/O port. The general
 * principle of operation is to check permissions for access to the I/O port at the time an IOPort
 * object is instantiated. In general, once instantiated, no additional permission checks are
 * required. The exception to this rule is that certain I/O ports span a range of addresses.
 * When accessing these kinds of I/O ports, it is necessary to perform a range check at run
 * time.
 */
public class IOPort {

    /**
     * No public constructors. Use the factory method instead.
     */
    protected @StaticAnalyzable IOPort();

    /**
     * Create an IOPort object to represent a single I/O address.
     *
     * Parameters:
     *
     * address: The hardware address of the physical I/O port.
     *
     * memory_mapped: If true, the address is mapped to the memory address space. If false,
     * the address is mapped to the I/O address space.
     *
     * port_width: Specifies how many bits to transfer on each read or write operation. Options
     * are 8, 16, 32, or 64.
     *
     * read_permission: If true, the application expects to be able to read values from the
     * requested IOPort object. Otherwise, the application does not expect to read from
     * the port.
     *
     * write_permission: If true, the application expects to write values to the requested IOPort

```

```
*      object. Otherwise, the application does not expect to write to the port.
*
* returns a reference to the created IOPort (or subclass) object
*
* throws EmbeddedConflictException if the target environment does not allow the requested
*      access to the specified address range
*/
final public static @StaticAnalyzable @CallerAllocatedResult IOPort
    createIOPort(long address, boolean memory_mapped, int port_width,
                boolean read_permission, boolean write_permission)
    throws EmbeddedConflictException;

/**
 * Create an IOPort object to represent a range of I/O addresses.
 *
 * Parameters:
 *
 * address: The hardware address of the start of the physical range of I/O ports.
 *
 * memory_mapped: If true, the address is mapped to the memory address space. If false,
 *      the address is mapped to the I/O address space.
 *
 * port_width: Specifies how many bits to transfer on each read or write operation. Options
 *      are 8, 16, 32, or 64.
 *
 * num_elements: The number of I/O ports to be represented by the requested IOPort
 *      object. These ports are presumed to be represented in contiguous memory, each
 *      port aligned on the next port_width bit boundary.
 *
 * read_permission: If true, the application expects to be able to read values from the
 *      requested IOPort object. Otherwise, the application does not expect to read from
 *      the port.
 *
 * write_permission: If true, the application expects to write values to the requested IOPort
 *      object. Otherwise, the application does not expect to write to the port.
 *
 * returns a reference to the created IOPort (or subclass) object.
 *
 * throws IllegalArgumentException if num_elements <= 0.
 *
 * throws EmbeddedConflictException if num_elements > 0 and the target environment does
 *      not allow the requested access to the specified address range.
 */
final public static @StaticAnalyzable @CallerAllocatedResult IOPort
    createIOPort(
        long address,
        boolean memory_mapped,
        int port_width,
        int num_elements,
        boolean read_permission,
        boolean write_permission)
    throws EmbeddedConflictException, IllegalArgumentException;
```

```
/**
 * Read a byte from this IOPort.
 *
 * returns the requested byte
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *         8-bit I/O, or if this object does not have read permission.
 */
public @StaticAnalyzable @ScopedThis byte readByte()
    throws UnsupportedOperationException;

/**
 * Write a byte to this IOPort.
 *
 * Parameters:
 *
 * b: the byte to be written.
 *
 * returns the written byte.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *         8-bit I/O, or if this object does not have write permission.
 */
public @StaticAnalyzable @ScopedThis byte writeByte(byte b)
    throws UnsupportedOperationException;

/**
 * Read a byte from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the
 *         first element. Offset N-1 is the Nth element.
 *
 * returns the requested byte.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *         8-bit I/O ports, or if this IOPort object does not have read permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *         in this IOPort object's range, or if offset is less than 0, but the operation would
 *         otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis byte readByte(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Write a byte to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *         first element. Offset N-1 is the Nth element.

```

```
*
* b: the byte to be written.
*
* returns the written byte.
*
* throws UnsupportedOperationException if this IOPort object does not represent a range of
*       8-bit I/O ports, or if this object does not have write permission.
*
* throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
*       in this IOPort object's range, or if offset is less than 0, but the operation would
*       otherwise be supported.
*/
public @StaticAnalyzable @ScopedThis byte writeByte(int offset, byte b)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Read a short from this IOPort.
 *
 * returns the requested short
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *       16-bit I/O, or if this object does not have read permission.
 */
public @StaticAnalyzable @ScopedThis short readShort()
    throws UnsupportedOperationException;

/**
 * Write a short to this IOPort.
 *
 * Parameters:
 *
 * s: the short to be written.
 *
 * returns the written short.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *       16-bit I/O, or if this object does not have write permission.
 */
public @StaticAnalyzable @ScopedThis short writeShort(short s)
    throws UnsupportedOperationException;

/**
 * Read a short from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the
 *       first element. Offset N-1 is the Nth element.
 *
 * returns the requested short.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range
 *       of 16-bit I/O ports, or if this IOPort object does not have read permission.
```

```
*
* throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
*       in this IOPort object's range, or if offset is less than 0, but the operation would
*       otherwise be supported.
*/
public @StaticAnalyzable @ScopedThis short readShort(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Write a short to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *       first element. Offset N-1 is the Nth element.
 *
 * s: the short to be written.
 *
 * returns the written short.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *       16-bit I/O ports, or if this object does not have write permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *       in this IOPort object's range, or if offset is less than 0, but the operation would
 *       otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis short writeShort(int offset, short s)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Read an int from this IOPort.
 *
 * returns the requested int
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *       32-bit I/O, or if this object does not have read permission.
 */
public @StaticAnalyzable @ScopedThis int readInt()
    throws UnsupportedOperationException;

/**
 * Write an int to this IOPort.
 *
 * Parameters:
 *
 * i: the int to be written.
 *
 * returns the written int.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *       32-bit I/O, or if this object does not have write permission.
 */
```

```
public @StaticAnalyzable @ScopedThis int writeInt(int i)
    throws UnsupportedOperationException;

/**
 * Read an int from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the
 *          first element. Offset N-1 is the Nth element.
 *
 * returns the requested int.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          32-bit I/O ports, or if this IOPort object does not have read permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis int readInt(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Write an int to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *          first element. Offset N-1 is the Nth element.
 *
 * i: the int to be written.
 *
 * returns the written int.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          32-bit I/O ports, or if this object does not have write permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis int writeInt(int offset, int i)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Read a float from this IOPort.
 *
 * returns the requested float
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *          32-bit I/O, or if this object does not have read permission.
 */
```

```
public @StaticAnalyzable @ScopedThis float readFloat()
    throws UnsupportedOperationException;

/**
 * Write a float to this IOPort.
 *
 * Parameters:
 *
 * f: the float to be written.
 *
 * returns the written float.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *         32-bit I/O, or if this object does not have write permission.
 */
public @StaticAnalyzable @ScopedThis float writeFloat(float f)
    throws UnsupportedOperationException;

/**
 * Read a float from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the
 *         first element. Offset N-1 is the Nth element.
 *
 * returns the requested float.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *         32-bit I/O ports, or if this IOPort object does not have read permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *         in this IOPort object's range, or if offset is less than 0, but the operation would
 *         otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis float readFloat(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Write a float to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *         first element. Offset N-1 is the Nth element.
 *
 * f: the float to be written.
 *
 * returns the written float.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *         32-bit I/O ports, or if this object does not have write permission.
 *
```

```
* throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
*      in this IOPort object's range, or if offset is less than 0, but the operation would
*      otherwise be supported.
*/
public @StaticAnalyzable @ScopedThis float writeFloat(int offset, float f)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Read a long from this IOPort.
 *
 * returns the requested long
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *      64-bit I/O, or if this object does not have read permission.
 */
public @StaticAnalyzable @ScopedThis long readLong()
    throws UnsupportedOperationException;

/**
 * Write a long to this IOPort.
 *
 * Parameters:
 *
 * I: the long to be written.
 *
 * returns the written long.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *      64-bit I/O, or if this object does not have write permission.
 */
public @StaticAnalyzable @ScopedThis long writeLong(long I)
    throws UnsupportedOperationException;

/**
 * Read a long from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the\
 *      first element. Offset N-1 is the Nth element.
 *
 * returns the requested long.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *      64-bit I/O ports, or if this IOPort object does not have read permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *      in this IOPort object's range, or if offset is less than 0, but the operation would
 *      otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis long readLong(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;
```

```
/**
 * Write a long to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *          first element. Offset N-1 is the Nth element.
 *
 * Parameters:
 *
 * l: the long to be written.
 *
 * returns the written long.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          64-bit I/O ports, or if this object does not have write permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis long writeLong(int offset, long l)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Read a double from this IOPort.
 *
 * returns the requested double
 *
 * throws UnsupportedOperationException if this IOPort object does not represent
 *          64-bit I/O, or if this object does not have read permission.
 */
public @StaticAnalyzable @ScopedThis double readDouble()
    throws UnsupportedOperationException;

/**
 * Write a double to this IOPort.
 *
 * Parameters:
 *
 * d: the double to be written.
 *
 * returns the written double.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          64-bit I/O ports, or if this object does not have write permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis double writeDouble(double d)
    throws UnsupportedOperationException;
```

```
/**
 * Read a double from this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be fetched. Offset 0 is the
 *          first element. Offset N-1 is the Nth element.
 *
 * returns the requested double.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          64-bit I/O ports, or if this IOPort object does not have read permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis double readDouble(int offset)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;

/**
 * Write a double to this IOPort.
 *
 * Parameters:
 *
 * offset: identifies which element of the IOPort subrange to be written. Offset 0 is the
 *          first element. Offset N-1 is the Nth element.
 *
 * d: the double to be written.
 *
 * returns the written double.
 *
 * throws UnsupportedOperationException if this IOPort object does not represent a range of
 *          64-bit I/O ports, or if this object does not have write permission.
 *
 * throws ArrayIndexOutOfBoundsException if offset is larger than the number of elements
 *          in this IOPort object's range, or if offset is less than 0, but the operation would
 *          otherwise be supported.
 */
public @StaticAnalyzable @ScopedThis double writeDouble(int offset, double d)
    throws UnsupportedOperationException, ArrayIndexOutOfBoundsException;
}
```

---

---

## IOPort16I.java

---

```
package javax.realtime.util.sc;

public class IOPort16I extends IOPort {
    public final short readShort();
    public final short readShort(int offset) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort16IO.java**

---

```
package javax.realtime.util.sc;

public class IOPort16IO extends IOPort {
    public final short readShort();
    public final short writeShort(short s);
    public final short readShort(int offset) throws ArrayIndexOutOfBoundsException;
    public final short writeShort(int offset, short s) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort16O.java**

---

```
package javax.realtime.util.sc;

public class IOPort16O extends IOPort {
    public final short writeShort(short s);
    public final short writeShort(int offset, short s) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort32I.java**

---

```
package javax.realtime.util.sc;

public class IOPort32I extends IOPort {
    public final int readInt();
    public final int readInt(int offset) throws ArrayIndexOutOfBoundsException;
    public final float readFloat();
    public final float readFloat(int offset) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort32IO.java**

---

```
package javax.realtime.util.sc;

public class IOPort32IO extends IOPort {
    public final int readInt();
    public final int writeInt(int i);
    public final int readInt(int offset);
    public final int writeInt(int offset, int i) throws ArrayIndexOutOfBoundsException;
    public final float readFloat();
    public final float writeFloat(float f);
    public final float readFloat(int offset) throws ArrayIndexOutOfBoundsException;
    public final float writeFloat(int offset, float f) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort32O.java**

---

```
package javax.realtime.util.sc;

public class IOPort32O extends IOPort {
    public final int writeInt(int i);
    public final int writeInt(int offset, int i) throws ArrayIndexOutOfBoundsException;
    public final float writeFloat(float f);
    public final float writeFloat(int offset, float f) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort64I.java**

---

```
package javax.realtime.util.sc;

public class IOPort64I extends IOPort {
    public final long readLong();
    public final long readLong(int offset) throws ArrayIndexOutOfBoundsException;
    public final double readDouble();
    public final double readDouble(int offset) throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort64IO.java**

---

```
package javax.realtime.util.sc;

public class IOPort64IO extends IOPort {
    public final long readLong();
    public final long writeLong(long l);
    public final long readLong(int offset) throws ArrayIndexOutOfBoundsException;
    public final long writeLong(int offset, long l) throws ArrayIndexOutOfBoundsException;
    public final double readDouble();
    public final double writeDouble(double d);
    public final double readDouble(int offset) throws ArrayIndexOutOfBoundsException;
    public final double writeDouble(int offset, double d)
        throws ArrayIndexOutOfBoundsException;
}
```

---

---

**IOPort64O.java**

---

```
package javax.realtime.util.sc;

public class IOPort64O extends IOPort {
    public final long writeLong(long l);
    public final long writeLong(int offset, long l) throws ArrayIndexOutOfBoundsException;
    public final double writeDouble(double d);
    public final double writeDouble(int offset, double d)
        throws ArrayIndexOutOfBoundsException;
}
```

```
}
```

---

---

### **IOPort8I.java**

---

```
package javax.realtime.util.sc;

public class IOPort8I extends IOPort {
    public final byte readByte();
    public final byte readByte(int offset) throws ArrayIndexOutOfBoundsException;
}
```

---

---

### **IOPort8IO.java**

---

```
package javax.realtime.util.sc;

public class IOPort8IO extends IOPort {
    public final byte readByte();
    public final byte writeByte(byte b);
    public final byte readByte(int offset) throws ArrayIndexOutOfBoundsException;
    public final byte writeByte(int offset, byte b) throws ArrayIndexOutOfBoundsException;
}
```

---

---

### **IOPort8O.java**

---

```
package javax.realtime.util.sc;

public class IOPort8O extends IOPort {
    public final byte writeByte(byte b);
    public final byte writeByte(int offset, byte b) throws ArrayIndexOutOfBoundsException;
}
```

---

---

### **ImmortalAllocation.java**

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;
import javax.realtime.util.sc.StaticLimit;

/**
 * Annotate a method or constructor with this annotation in order to indicate that the method
 * may allocate ImmortalMemory. Any method that might allocate ImmortalMemory cannot be
 * invoked from a method that may not allocate ImmortalMemory.
 *
 * Note that this annotation is not inherited.
 */
@Documented @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME) public @interface ImmortalAllocation {
}
```

---

---

## InitializeAtStartup.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation is associated with a static class variable. It signifies that the static
 * initialization code associated with this class variable cannot be performed at link time.
 * Rather, the initialization code must be executed at run time during startup, before execution
 * of any application code.
 */
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
@Target({ElementType.FIELD}) public @interface InitializeAtStartup {
}
```

---

---

## InterruptEvent.java

---

```
package javax.realtime.util.sc;

/**
 * This class generalizes on what's already available in RTSJ. InterruptEvent has a queue size of
 * "zero". Every interrupt must be serviced "immediately". We don't maintain a queue, but we do
 * timestamp the firing.
 *
 * In case we miss the deadline, the miss_handler associated with the InterruptHandler's
 * associated ReleaseParameters will be asynchronously fired. Note that the fire() invocation is
 * triggered from within the epilogue code associated with execution of the corresponding
 * InterruptHandler.handleAsyncEvent() code.
 */
public class InterruptEvent extends AsyncEvent {

    /**
     * Construct a new InterruptEvent object
     *
     * Parameters:
     *
     * interrupt_no: the system-dependent number that uniquely identifies the physical interrupt
     * with which we desire to associate this object.
     *
     * handler: the InterruptHandler object that represents the code to be executed each time
     * this interrupt is triggered by hardware.
     *
     * throws EmbeddedConflictException if the interrupt number is not available, or permission is
     * not granted to this code to latch onto this interrupt number, or if there is
     * incompatibility between handler and this interrupt_no (e.g. this interrupt runs at a
     * different priority than what was specified for handler).
     */
    public @StaticAnalyzable @ScopedPure
    InterruptEvent(int interrupt_no, InterruptHandler handler)
    throws EmbeddedConflictException;
```

```

/**
 * When an InterruptEvent is constructed, it is initially in unarmed status, meaning that
 * hardware interrupts will be ignored by this InterruptEvent object. Invoke arm() to enable
 * future hardware triggerings of this interrupt to cause the corresponding
 * InterruptEventHandler to execute its asyncEventHandler() code.
 */
public @StaticAnalyzable @ScopedThis void arm();

/**
 * Invoke disarm() to cause future hardware triggerings of this object's associated interrupt
 * to be ignored until a subsequent invocation of the arm() method.
 */
public @StaticAnalyzable @ScopedThis void disarm();

/**
 * This potentially (@Scoped) interrupt event is armed and left armed
 * until the potentially (@Scoped) Runnable code r returns, at which point
 * this InterruptEvent is disarmed. This is the only way to arm a @Scoped
 * InterruptEvent object.
 *
 * r: the code that runs while we maintain this InterruptEvent in armed status. Upon return
 * from r, we automatically disarm this InterruptEvent object.
 */
@ScopedPure public void runWhileArmed(Runnable r);
}

```

---



---

## InterruptHandler.java

---

```

package javax.realtime.util.sc;

import javax.realtime.MemoryArea;
import javax.realtime.MemoryParameters;
import javax.realtime.SchedulingParameters;

/**
 * This class generalizes what's already available in RTSJ.
 * <p>
 * When we instantiate one of these objects, we also instantiate a special form of thread to
 * serve as its bound asynchronous event handler. The corresponding thread data structure is
 * non-traditional in the sense that this thread is never scheduled by software and it never
 * needs to block. What's most important is to assure that stack memory is reserved for
 * execution of the interrupt handler and this stack memory is part of the affiliated thread's
 * data structure.
 *
 * Notes:
 *
 * ♦ We don't allow the noheap versions of the constructor, because those versions imply that
 * we have a choice, and we don't...
 *
 * ♦ Also, we don't provide addToFeasibility() because that implies we are doing a feasibility
 * analysis, and we are not

```

```
*/
public abstract class InterruptHandler extends BoundAsyncEventHandler implements Atomic {
/**
 * Instantiate an InterruptHandler object and its corresponding
 * thread data structure.
 *
 * Parameters:
 *
 * scheduling: A SchedulingParameters object which will be associated with the constructed
 * instance. This must be non-null. Note that the only scheduling supported by the
 * safety-critical profile is fixed-priority without time slicing. This argument must be
 * an instance of PriorityParameters and the corresponding value of the priority field
 * must be in the range associated with interrupt handling.
 *
 * release: A ReleaseParameters object which will be associated with the constructed
 * instance. This must be non-null and it must be an instance of AperiodicParameters or
 * SporadicParameters. The queue length must equal one. The queue overflow behavior
 * must equal ReleaseParameters.arrivalTimeQueueOverflowIgnore.
 *
 * memory: A MemoryParameters object which will be associated with the constructed
 * instance. This must be non-null. We require MemoryParameters.getMaxImmortal()
 * to return zero.
 *
 * is_daemon: If true, marks this event handler as a daemon event handler. If false, marks
 * this event handler as a user event handler. The Real-Time Virtual Machine exits when
 * the only schedulable objects and threads running are all daemons.
 *
 * throws IllegalArgumentException if priority set to interrupt level or is otherwise outside
 * the range of legal priority levels.
 *
 * throws NullPointerException if any of its arguments equal null.
 *
 * throws OutOfMemoryError exception if there isn't enough memory to instantiate both this
 * object and the corresponding thread.
 *
 * throws IllegalArgumentException If scheduling or release or memory or area are
 * incompatible with our intent to execute this code as a hardware-driven interrupt
 * handler.
 */
public @StaticAnalyzable @ScopedPure InterruptHandler(SchedulingParameters scheduling,
    ReleaseParameters release, MemoryParameters memory, boolean is_daemon)
    throws NullPointerException, OutOfMemoryError, IllegalArgumentException;

// See superclass documentation.
public @StaticAnalyzable @ScopedThis int getAndClearPendingFireCount();

// See superclass documentation
public @StaticAnalyzable @ScopedThis int getAndDecrementPendingFireCount();

// See superclass documentation
public @StaticAnalyzable @ScopedThis int getAndIncrementPendingFireCount();

// See superclass documentation
```

(Permission granted to reproduce and distribute as a complete document, without modification)

```

public @StaticAnalyzable @ScopedThis @Scoped MemoryArea getMemoryArea();

// See superclass documentation
public @StaticAnalyzable @ScopedThis int getPendingFireCount();

// See superclass documentation
public @StaticAnalyzable @ScopedThis @Scoped
    MemoryParameters getMemoryParameters();

// See superclass documentation
public @StaticAnalyzable @ScopedThis @Scoped
    ReleaseParameters getReleaseParameters();

// See superclass documentation
public @StaticAnalyzable @ScopedThis @Scoped
    SchedulingParameters getSchedulingParameters();

/**
 * Since this class is declared with the StaticAnalyzable annotation, any implementation of
 * this method must be execution-time analyzable. Further, we require that the byte-code
 * verifier enforce that StaticAnalyzable's heap_bytes attribute equals zero.
 *
 * Execution of the interrupt handling code always occurs at the interrupt-level priority.
 *
 * On a uniprocessor system, while this object's handleAsyncEvent() is running, no lower
 * priority thread may request access to data structures protected by a priority ceiling lock
 * with ceiling equal to this thread's priority. On the other hand, if some thread has acquired
 * a priority ceiling lock with ceiling priority equal to or higher than this object's priority, and
 * a hardware interrupt is fired, the interrupt handling will be deferred until that thread
 * releases the lock and its priority lowers below the interrupt handler's priority.
 *
 * On a multiprocessor system, it is conceivable that multiple interrupt-level threads might be
 * running at the same priority in parallel. Thus, enforcing mutual exclusion with interrupt
 * handlers requires more than simply elevating an application thread's priority to interrupt
 * level. It is necessary for the application thread and the interrupt handler to synchronize on
 * a shared priority ceiling lock object. Note further that proper use of Java's built-in
 * synchronized statements is also essential for enforcing compiler-controlled data coherency
 * (copying registers to memory and explicitly committing cache contents to memory). In
 * summary, programmers who need to share data between an interrupt handler and
 * application threads should generally use a priority ceiling emulation synchronization lock to
 * enforce mutual exclusion.
 *
 * The application overrides this method to provide its own interrupt handler.
 */
public @StaticAnalyzable @ScopedThis abstract void handleAsyncEvent();
}

```

---

## NoHeapRealtimeThread.java

---

```

package javax.realtime.util.sc;

import javax.realtime.MemoryArea;

```

(Permission granted to reproduce and distribute as a complete document, without modification)

```

import javax.realtime.SchedulingParameters;

public abstract class NoHeapRealtimeThread extends javax.realtime.RealtimeThread {

    /**
     * Create a real-time thread with the given characteristics. This is the same as
     * javax.realtime.NoHeapRealtimeThread(scheduling, null, area).
     *
     * mode: Specifies whether the scheduling of this thread should be handled in an
     * implementation-defined way by the underlying real-time operating system (NATIVE),
     * or in full compliance with the established standards for scheduling of hard real-time
     * threads (COMPLIANT). Only NATIVE threads are allowed to invoke native methods.
     *
     * scheduling: The SchedulingParameters associated with this (and possibly other instances
     * of Schedulable). If scheduling is null, the default is a copy of the creator's
     * scheduling parameters created in the same memory area as the new
     * NoHeapRealtimeThread.
     *
     * area: The MemoryArea associated with this. If area is null,
     * java.lang.IllegalArgumentException is thrown.
     *
     * throws IllegalArgumentException thrown if the parameters are not compatible with the
     * default scheduler, or if area is null.
     */
    public @StaticAnalyzable @ScopedPure NoHeapRealtimeThread(ThreadModes mode,
        SchedulingParameters scheduling, MemoryArea area)
        throws java.lang.IllegalArgumentException;

    /**
     * Create a real-time thread with the given characteristics. This is the same as
     * javax.realtime.NoHeapRealtimeThread(null, scheduling, release, null, area).
     *
     * mode: Specifies whether the scheduling of this thread should be handled in an
     * implementation-defined way by the underlying real-time operating system (NATIVE),
     * or in full compliance with the established standards for scheduling of hard real-time
     * threads (COMPLIANT). Only NATIVE threads are allowed to invoke native methods.
     *
     * scheduling: The SchedulingParameters associated with this (and possibly other instances of
     * Schedulable). If scheduling is null, the default is a copy of the creator's scheduling
     * parameters created in the same memory area as the new NoHeapRealtimeThread.
     *
     * release: The ReleaseParameters associated with this (and possibly other instances of
     * Schedulable ). If release is null, it defaults to a copy of the creator's release
     * parameters created in the same memory area as the new NoHeapRealtimeThread.
     *
     * area: The MemoryArea associated with this. If area is null,
     * java.lang.IllegalArgumentException is thrown.
     *
     * throws IllegalArgumentException if the parameters are not compatible with the default
     * scheduler, or if area is null.
     */
    public @StaticAnalyzable @ScopedPure
        NoHeapRealtimeThread(ThreadModes mode, SchedulingParameters scheduling,

```

```

        ReleaseParameters release, MemoryArea area)
    throws java.lang.IllegalArgumentException;

/**
 * A NATIVE-mode thread calls transfigure() to temporarily turn itself into a COMPLIANT
 * thread. A COMPLIANT thread is scheduled by the hard real-time Java run-time
 * environment, honoring priority and synchronization semantics described in these guidelines.
 */
public static void transfigure();

/**
 * A transfigured NATIVE-mode thread calls condescend() to restore itself to NATIVE mode.
 * A NATIVE thread is scheduled by the underlying (real-time) operating system, implementing
 * semantics that may differ from one environment to the next.
 */
public static void condescend();

public enum ThreadModes {
    NATIVE,          // implemented by the underlying operating system providing semantics
                    // that are implementation defined. Only NATIVE threads are allowed to
                    // invoke arbitrary native methods (which might block).
    COMPLIANT,      // implemented in full compliance with the official standard for
                    // priorities, synchronization queues, etc. Note that COMPLIANT may be
                    // implemented either by green threads, or by the underlying RTOS. We
                    // don't care. We do, however, restrict that COMPLIANT threads cannot
                    // invoke undisciplined native methods because we expect that some
                    // implementations of COMPLIANT threads will need to be implemented by
                    // green threads.
    TRANSFIGURED_NATIVE,
                    // A transfigured-native thread is a NATIVE thread that is currently
                    // executing under the control of the hard real-time scheduler.
    JAVA,           // A JAVA thread is a traditional Java thread running within the hard
                    // real-time environment (executing @TraditionalJavaMethod methods)
                    // under the control of the traditional Java VM's scheduler.
    TRANSFIGURED_JAVA
                    // A transfigured-java thread is a traditional Java thread executing
                    // @TraditionalJavaMethod methods under control of the hard real-time
                    // environment's scheduler. A JAVA thread automatically transfigures
                    // itself upon first attempt to synchronize or block within the hard
                    // real-time environment. A transfigured-Java thread automatically
                    // condescends upon departing all synchronization contexts.
};

/**
 * Returns the mode of this thread
 */
public @StaticAnalyzable @ScopedThis ThreadModes mode();

public enum ThreadStates {
    GESTATING,          // thread has not yet been started
    NATIVE_SCHEDULED,   // thread's mode is NATIVE and
                        // scheduling of the thread has been

```

```

        // turned over to the underlying OS.
        // For NATIVE threads, we cannot
        // determine whether the thread is
        // RUNNING or READY, so we simply
        // return NATIVE_SCHEDULED status.
    RUNNING, // thread is currently running
    READY, // thread is ready to run
    AWAITING_ACTIVATION, // thread is driven by an async event
        // handler, and is waiting to be
        // activated
    WAITING, // thread is waiting in object.wait()
        // - not possible in safety-critical
        // profile
    SYNC_BLOCKED, // thread is waiting on synchronized
        // statement or method - not possible
        // in safety-critical profile
    MUTEX_BLOCKED, // thread is waiting to acquire mutex
        // - not possible in safety-critical
        // profile
    SEM_BLOCKED, // thread is waiting to acquire
        // semaphore P() - not possible in
        // safety-critical profile
};

/**
 * Returns the status of this thread,
 */
public @StaticAnalyzable @ScopedThis ThreadStates status();
}

```

---

## OneShotTimer.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

/**
 * This class mimics the behavior of javax.realtime.OneShotTimer.
 * <p>
 * RTSJ additions:
 * ♦ restart();
 * ♦ cancel();
 *
 * RTSJ refinements:
 * ♦ We specify the following portability semantics:
 *
 * ♦ Each timer is associated with a particular clock. That clock has a particular tick
 * interval. The first tick following a particular event is assumed to represent passage
 * of zero time. The second tick is assumed to represent passage of one complete time
 * tick, and so on. When a timer is set for a specified time, the event trigger is rounded
 * down to the nearest tick preceding the desired target time. Note that every
 * scheduled event will wait for at least one tick.

```

- \*
  - \* ♦ No memory will be allocated when a `OneShotTimer` is restarted.
  - \*
    - \* ♦ The time complexity of constructing a new `OneShotTimer` is logarithmic in the number of timers previously set and not yet expired for the associated `Clock` object.
    - \*
      - \* ♦ The time complexity of canceling a previously set timer is constant.
      - \*
        - \* ♦ The time complexity of restarting an existing `OneShotTimer` is logarithmic in the number of timers previously set and not yet expired for the associated `Clock` object.
        - \*
          - \* ♦ A timer that is canceled (or restarted) before it expired is considered to be unexpired for purposes of the complexity analysis that is mentioned in the descriptions of the constructor and the `restart()` operation.

```

*/
public class OneShotTimer extends javax.realtime.OneShotTimer {

    /**
     * Create a OneShotTimer instance, which is a subclass of AsyncEvent, based on the Clock
     * associated with the time parameter, that will trigger execution of the handler's
     * handleAsyncEvent() method at the appropriate time.
     *
     * Parameters:
     *
     * time: The time at which the handler is made ready to run. A null value of time is equivalent
     * to a relative time of 0.
     *
     * handler: The AsyncEventHandler that will be scheduled at the specified time.
     *
     * throws IllegalArgumentException if time is a RelativeTime instance less than zero or if the
     * Clock associated with time is not an active Clock.
     */
    public @ScopedPure OneShotTimer(HighResolutionTime time, AsyncEventHandler handler)
        throws IllegalArgumentException;

    /**
     * Create a OneShotTimer instance, which is a subclass of AsyncEvent, based on the Clock
     * associated with the time parameter, that will trigger execution of the handler's
     * handleAsyncEvent() method at the appropriate time.
     *
     * Parameters:
     *
     * time: The time at which the handler is made ready to run. A null value of time is equivalent
     * to a relative time of 0.
     *
     * handler: The AsyncEventHandler that will be scheduled at the specified time.
     *
     * throws IllegalArgumentException if time is a RelativeTime instance less than zero or if
     * the Clock associated with time is not an active clock.
     */
    public @ScopedPure OneShotTimer(HighResolutionTime time, AsyncEventHandler handlers[])
        throws IllegalArgumentException;

```

```
/**
 * Cancel a previously set timer. This is different than disable, because disable leaves the
 * timer in the queue of pending events, and if a disabled timer is enabled before the target
 * time has been reached, the corresponding event will still be triggered for execution.
 */
public @StaticAnalyzable void cancel();

/**
 * If the target time is represented as an AbsoluteTime object, restart() has only the effect
 * of making this Timer active and enabled if it wasn't already active or if it wasn't already
 * enabled. If the target time is represented as a RelativeTime object, restart() also has the
 * effect of changing the absolute target time to NOW plus the value of the RelativeTime
 * value associated with this timer.
 *
 * We require that the time complexity of this method's implementation be logarithmic in the
 * number of Timer events previously associated with the same clock and not yet expired.
 *
 * With respect to "atomicity", we expect restart() to behave the same as reschedule().
 *
 * throws IllegalStateException if the target time is represented by an AbsoluteTime object
 * that has already passed.
 */
public void restart() throws IllegalStateException;
}
```

---

## PCP.java

---

```
package javax.realtime.util.sc;

/**
 * For any class that implements PCP, all synchronization locking is performed using the
 * immediate priority ceiling protocol.
 */
public interface PCP {

    /**
     * The programmer who implements PCP is required to provide an implementation of
     * ceilingPriority(). At the time a PCP object is instantiated, the system checks the current
     * MonitorControl policy. If the MonitorControl policy does not equal PriorityCeilingEmulation,
     * or the the value returned from the current MonitorControl policy's getCeiling() method does
     * not equal the value returned from this method, the object is not instantiated and the
     * constructor throws an IllegalStateException.
     *
     * In situations for which it is important to statically determine ceiling priorities of particular
     * classes, programmers can annotate the ceilingPriority() method with a @Ceiling(int)
     * annotation. The value attribute of this annotation represents the intended ceiling priority of
     * instances of this class. Note that the argument to the @Ceiling() annotation may be an
     * arithmetic expression which includes symbolic constants.
     */
    public @StaticAnalyzable @ScopedThis int ceilingPriority();
}
```

---



---

**PeriodicParameters.java**


---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

public class PeriodicParameters extends javax.realtime.PeriodicParameters {

    /**
     * Create a PeriodicParameters object. The queue overflow behavior is set to the default value
     * of arrivalTimeQueueOverflowIgnore and the queue length is set to one.
     *
     * Parameters:
     *
     * start: Time at which the first release begins (i.e. the schedulable object becomes eligible
     * for execution. If a RelativeTime, this time is relative to the first time the schedulable
     * object becomes activated (that is, when start() is called on a real-time thread or when an
     * associated event handler is first fired). If an AbsoluteTime, then the first release is
     * the maximum of the start parameter and the call to the associated RealtimeThread.start()
     * method (or the first firing of an associated event). If null, the default value is a new
     * instance of RelativeTime(0,0).
     *
     * period: The period is the interval between successive periods. There is no default value. If
     * period is null an exception is thrown.
     *
     * deadline: The latest permissible completion time measured from the release time of the
     * associated invocation of the schedulable object. Non-priority schedulers may use
     * this parameter to compute execution eligibility. If null, the default value is new
     * instance of RelativeTime(period).
     *
     * miss_handler: This handler is invoked if the run() method of the schedulable object is still
     * executing after the deadline has passed. Although minimum implementations do not
     * consider deadlines in feasibility calculations, they must recognize variable deadlines
     * and invoke the miss handler as appropriate. If null, the default value no deadline miss
     * handler.
     *
     * throws java.lang.IllegalArgumentException if the period is null or its time value is not
     * greater than zero, or if the time value of cost is less than zero, or if the time value
     * of deadline is not greater than zero.
     */
    public @StaticAnalyzable @ScopedPure
        PeriodicParameters(HighResolutionTime start, RelativeTime period,
            RelativeTime deadline, AsyncEventHandler miss_handler);

    /**
     * Create a PeriodicParameters object.
     *
     * Parameters:
     *
     * start: Time at which the first release begins (i.e. the schedulable object becomes eligible
     * for execution. If a RelativeTime, this time is relative to the first time the
     * schedulable object becomes activated (that is, when start() is called on a real-time

```

```

*      thread or when an associated event handler is first fired). If an AbsoluteTime, then
*      the first release is the maximum of the start parameter and the call to the
*      associated RealtimeThread.start() method (or the first firing of an associated
*      event). If null, the default value is a new instance of RelativeTime(0,0).
*
* period: The period is the interval between successive periods. There is no default value. If
*      period is null an exception is thrown.
*
* deadline: The latest permissible completion time measured from the release time of the
*      associated invocation of the schedulable object. Non-priority schedulers may use
*      this parameter to compute execution eligibility. If null, the default value is new
*      instance of RelativeTime(period).
*
* miss_handler: This handler is invoked if the run() method of the schedulable object is still
*      executing after the deadline has passed. Although minimum implementations do not
*      consider deadlines in feasibility calculations, they must recognize variable deadlines
*      and invoke the miss handler as appropriate. If null, the default value no deadline miss
*      handler.
*
* queue_length: The length of the arrival-time queue. This queue is used for the purpose of
*      enforcing compliance with deadlines.
*
* queue_overflow_behavior: The String encoding of the desired behavior in the case that the
*      arrival-time queue overflows. The value of this argument should be either
*      ReleaseParameters.arrivalTimeQueueOverflowIgnore or
*      ReleaseParameters.arrivalTimeQueueOverflowReplace.
*
* throws java.lang.IllegalArgumentException if the period is null or its time value is not
*      greater than zero, or if the time value of cost is less than zero, or if the time value
*      of deadline is not greater than zero, or if queue_length is less than one or if
*      queue_overflow_behavior does not equal
*      ReleaseParameters.arrivalTimeQueueOverflowIgnore or
*      ReleaseParameters.arrivalTimeQueueOverflowReplace.
*/
public @StaticAnalyzable @ScopedPure
    PeriodicParameters(HighResolutionTime start, RelativeTime period, RelativeTime deadline,
        AsyncEventHandler miss_handler, int queue_length, String queue_overflow_behavior);

/**
 * Gets a reference to the arrival_time_queue_overflow_behavior configuration.
 *
 * returns a reference to the value of arrival_time_queue_overflow_behavior.
 */
public @StaticAnalyzable @ScopedThis String getArrivalTimeQueueOverflowBehavior();

/**
 * Gets the initial (and final, because this state variable cannot change) number of entries that
 * the arrival time queue can hold.
 *
 * returns the length of the arrival-time queue.
 */
public @StaticAnalyzable @ScopedThis int getInitialArrivalTimeQueueLength();
}

```

(Permission granted to reproduce and distribute as a complete document, without modification)

---

---

**PeriodicTimer.java**

---

```
package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

public class PeriodicTimer extends javax.realtime.PeriodicTimer {

    /**
     * Create an instance of AsyncEvent that executes its fire() method periodically. The time
     * complexity of this method's implementation shall be proportional to the size of the handlers
     * array.
     *
     * Parameters:
     *
     * start: The time that specifies when the first interval begins, based on the clock associated
     * with it. If start is null, then the first interval will start immediately.
     *
     * interval: The period of the timer. Its usage is based on the clock associated with it.
     *
     * handler: The object that will be released when the timer fires.
     *
     * throws java.lang.IllegalArgumentException if start or interval or handler is null, or if any
     * element of the handlers array equals null, or if start is a RelativeTime instance with a
     * value less than zero, or if interval is a RelativeTime instance with a value less than or
     * equal to zero, or if interval and handler are associated with difference Clock objects,
     * or if the Clock associated with interval is not an active Clock.
     */
    public @ScopedPure PeriodicTimer(HighResolutionTime start, RelativeTime interval,
        AsyncEventHandler handler)
        throws IllegalArgumentException;

    /**
     * Create an instance of AsyncEvent that executes its fire() method periodically. The time
     * complexity of this method's implementation shall be proportional to the size of the handlers
     * array.
     *
     * Parameters:
     *
     * start: The time that specifies when the first interval begins, based on the clock associated
     * with it. If start is null, then the first interval will start immediately.
     *
     * interval: The period of the timer. Its usage is based on the clock associated with it.
     *
     * handlers: The array of AsyncEventHandler objects that will be released when the timer
     * fires.
     *
     * throws java.lang.IllegalArgumentException if start or interval or handler is null, or if any
     * element of the handlers array equals null, or if start is a RelativeTime instance with a
     * value less than zero, or if interval is a RelativeTime instance with a value less than or
     * equal to zero, or if interval and handler are associated with difference Clock objects,
     * or if the Clock associated with interval is not an active Clock.
     */
}
```

```

*/
public @ScopedPure PeriodicTimer(HighResolutionTime start, RelativeTime interval,
    @ScopedArray AsyncEventHandler handlers [])
    throws IllegalArgumentException;

/**
 * Gets the start time of this Timer, in a newly allocated instance of AbsoluteTime or
 * RelativeTime, depending on the type of the start value provided at the time this object was
 * instantiated.
 *
 * returns a new instance with the value of this timer's start time.
 */
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    HighResolutionTime getStartTime();
}

```

---



---

## PreallocatedExceptions.java

---

```

package javax.realtime.util.sc;

import javax.realtime.ArrivalTimeQueueOverflowException;
import javax.realtime.MITViolationException;
import javax.realtime.UnknownHappeningException;

/**
 * This class maintains references to each of the primordial preallocated shared instances of
 * each of the "standard" Throwable classes.
 */
public class PreallocatedExceptions {

    // Exceptions from javax.realtime package

    public static final ArrivalTimeQueueOverflowException
        ArrivalTimeQueueOverflowException = new ArrivalTimeQueueOverflowException();

    public static final MITViolationException
        MITViolationException = new MITViolationException();

    public static final UnknownHappeningException
        UnknownHappeningException = new UnknownHappeningException();

    // Exceptions from javax.realtime.util.sc package

    public static final EmbeddedConflictException
        EmbeddedConflictException = new EmbeddedConflictException();

    public static final UnsignedCoercionException
        UnsignedCoercionException = new UnsignedCoercionException();

    // Exceptions from java.lang package

```

```
public static final ArithmeticException
    ArithmeticException = new ArithmeticException();

public static final ArrayIndexOutOfBoundsException
    ArrayIndexOutOfBoundsException = new ArrayIndexOutOfBoundsException();

public static final ClassCastException
    ClassCastException = new ClassCastException();

public static final ClassFormatError ClassFormatError = new ClassFormatError();

public static final ClassNotFoundException
    ClassNotFoundException = new ClassNotFoundException();

public static final Error Error = new Error();

public static final Exception Exception = new Exception();

public static final IllegalArgumentException
    IllegalArgumentException = new IllegalArgumentException();

public static final IllegalMonitorStateException
    IllegalMonitorStateException = new IllegalMonitorStateException();

public static final IllegalStateException
    IllegalStateException = new IllegalStateException();

public static final IndexOutOfBoundsException
    IndexOutOfBoundsException = new IndexOutOfBoundsException();

public static final InstantiationException
    InstantiationException = new InstantiationException();

public static final InterruptedException
    InterruptedException = new InterruptedException();

public static final LinkageError LinkageError = new LinkageError();

public static final NegativeArraySizeException
    NegativeArraySizeException = new NegativeArraySizeException();

public static final NoSuchMethodException
    NoSuchMethodException = new NoSuchMethodException();

public static final NullPointerException
    NullPointerException = new NullPointerException();

public static final OutOfMemoryError OutOfMemoryError = new OutOfMemoryError();

public static final RuntimeException RuntimeException = new RuntimeException();

public static final SecurityException SecurityException = new SecurityException();
```

**(Permission granted to reproduce and distribute as a complete document, without modification)**

```
public static final UndeclaredThrowableException
    UndeclaredThrowableException = new UndeclaredThrowableException();

public static final UnsupportedOperationException
    UnsupportedOperationException = new UnsupportedOperationException();

}
```

---

## RelativeTime.java

---

```
package javax.realtime.util.sc;

/**
 * This class mimics javax.realtime.RelativeTime.
 * <p>
 * RTSJ additions:
 * ♦ UNDEFINED_TIME
 * ♦ RelativeTime(TimeBuffer, Clock)
 * ♦ RelativeTime(TimeBuffer);
 * ♦ subtract(long millis, int nanos)
 */
public class RelativeTime extends javax.realtime.RelativeTime {

    /**
     * Maintain a single immutable RelativeTime instance residing in ImmortalMemory which
     * represents UNDEFINED_TIME.
     */
    public static final RelativeTime UNDEFINED_TIME;

    /**
     * Construct a RelativeTime object representing an interval based on the parameter millis plus
     * the parameter nanos. The construction is subject to millis and nanos parameters
     * normalization. If there is an overflow in the millisecond component when normalizing then
     * an ArithmeticOverflowException will be thrown.
     *
     * The clock association is implicitly made with javax.realtime.util.sc.Clock.getTimeKeeper().
     *
     * Parameters:
     *
     * millis: The desired value for the millisecond component of this. The actual value is the
     * result of parameter normalization.
     *
     * nanos: The desired value for the nanosecond component of this. The actual value is the
     * result of parameter normalization.
     *
     * throws java.lang.ArithmeticOverflowException if there is an overflow in the millisecond
     * component when normalizing. Note that the RTSJ says this throws
     * IllegalArgumentException on overflow during normalization, but everywhere else,
     * they throw ArithmeticOverflowException in this situation. Assume the RTSJ is in
     * error.
     */
}
```

```
public @StaticAnalyzable @ScopedThis RelativeTime(long millis, int nanos)
    throws ArithmeticException;

/**
 * Construct a RelativeTime object representing an interval based on the parameter millis plus
 * the parameter nanos. The construction is subject to millis and nanos parameters'
 * normalization. If there is an overflow in the millisecond component when normalizing then
 * an ArithmeticOverflowException will be thrown.
 *
 * The constructed RelativeTime object is associated with the Clock represented by the
 * constructor's clock argument.
 *
 * Parameters:
 *
 * millis: The desired value for the millisecond component of this. The actual value is the
 * result of parameter normalization.
 *
 * nanos: The desired value for the nanosecond component of this. The actual value is the
 * result of parameter normalization.
 *
 * clock: The javax.realtime.util.sc.Clock object with which to associate this relative time
 * object.
 *
 * throws java.lang.ArithmeticOverflowException if there is an overflow in the millisecond
 * component when normalizing. Note that the RTSJ says this throws
 * IllegalArgumentException on overflow during normalization, but everywhere else,
 * they throw ArithmeticOverflowException in this situation. Assume the RTSJ is in
 * error.
 */
public @StaticAnalyzable @ScopedPure RelativeTime(long millis, int nanos, Clock clock)
    throws ArithmeticException;

/**
 * Make a new RelativeTime object from the given RelativeTime argument. The clock
 * association is made with the clock parameter.
 *
 * Parameters:
 *
 * time: The RelativeTime object which is the source for the copy.
 *
 * clock: The clock providing the association for the newly constructed object.
 *
 * throws java.lang.IllegalArgumentException if the time or * clock parameters are null.
 */
public @StaticAnalyzable @ScopedPure RelativeTime(RelativeTime time, Clock clock);

/**
 * Make a new RelativeTime object from the given TimeBuffer argument. The clock
 * association is implicitly made with the javax.realtime.util.sc.Clock.getTimeKeeper().
 *
 * Parameters:
 *
 * buffer: The TimeBuffer object to be converted.
```

```
*
* throws java.lang.IllegalArgumentException if the time parameter is null.
*/
public @StaticAnalyzable @ScopedPure RelativeTime(TimeBuffer buffer)
    throws IllegalArgumentException;

/**
 * Make a new RelativeTime object from the given TimeBuffer argument. The clock
 * association is made with the clock parameter.
 *
 * Parameters:
 *
 * buffer: The TimeBuffer object to be converted.
 *
 * throws java.lang.IllegalArgumentException if the time or clock parameters are null.
 */
public @StaticAnalyzable @ScopedPure RelativeTime(TimeBuffer buffer, Clock clock)
    throws IllegalArgumentException;

/**
 * Create a new object representing the result of subtracting millis and nanos from the values
 * of this and normalizing the result. The result will have the same clock association as this.
 * An ArithmeticException is thrown if there is an overflow in the result after normalization.
 *
 * Note that this method is missing from RTSJ - not sure why.
 *
 * Parameters:
 *
 * millis: The number of milliseconds to be added to this.
 *
 * nanos: The number of nanoseconds to be added to this.
 *
 * returns a new RelativeTime object whose time is the normalization of this plus millis
 * and nanos.
 *
 * throws java.lang.ArithmeticException if there is an overflow in the result after
 * normalization.
 */
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    RelativeTime subtract(long millis, int nanos)
        throws IllegalArgumentException, ArithmeticException;

/**
 * This method is different than the RTSJ in that it does not expect a Clock argument. In
 * general, we would expect any automatic conversion from an existing HighResolutionTime to
 * an AbsoluteTime would need to be associated with the same Clock. Assuming that we want
 * to enforce this restriction, the Clock argument is redundant.
 *
 * Convert the time of this to an absolute time. The returned AbsoluteTime object will be
 * associated with the same Clock as this. See the derived class comments for more
 * specific information.
 *
 * returns the AbsoluteTime conversion in a newly allocated object.

```

```

*/
public @StaticAnalyzable @ScopedThis @CallerAllocatedResult AbsoluteTime absolute();

/**
 * This method is defined differently than in RTSJ. In the RTSJ, this method takes a clock
 * argument and creates a RelativeTime associated with that clock argument which represents
 * the difference between this and the current time. Though not detailed in the RTSJ
 * description of this method, in order to calculate the difference, it is necessary to first
 * convert this into a time representation that is consistent with the supplied clock argument.
 * The behavior of this method within the RTSJ is inconsistent with other AbsoluteTime
 * difference methods, which are generally defined to throw an IllegalArgumentException if
 * the clocks associated with the two objects to be compared are different.
 *
 * So the suggested API here is to remove the clock argument and require that the returned
 * RelativeTime result is associated with the same clock as this AbsoluteTime object.
 *
 * Convert the time of this to a relative time by subtracting the current time from this time.
 * A destination object is allocated to return the result. The clock associated with the result
 * is the same clock that is associated with this object.
 *
 * In contexts for which the returned RelativeTime object is to be stack allocated, it is the
 * programmer's responsibility to assure that the associated clock's lifetime is at least as long
 * as this newly constructed object. Since there are no APIs that would allow creation of an
 * AbsoluteTime object with a reference to a @Scoped clock, we are assured within the
 * safety-critical profile that the clock resides in ImmortalMemory, so this requirement is
 * automatically satisfied.
 *
 * returns the RelativeTime conversion results in a newly allocated object associated with the
 * same clock as this.
 */
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis RelativeTime relative();
}

```

---

## ReleaseParameters.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

/**
 * This class mimics javax.realtime.ReleaseParameters.
 *
 * RTSJ additions:
 * ♦ ReleaseParameters(deadline, miss_handler);
 * ♦ ReleaseParameters(deadline, miss_handler, queue_length, overflow_behavior);
 * ♦ getArrivalTimeQueueOverflowBehavior();
 * ♦ getInitialArrivalTimeQueueLength();
 *
 * RTSJ refinements:
 *
 * ♦ In the RTSJ spec, arrival-time queue overflow behavior is associated only with
 * SporadicParameters. But that's not reasonable, because there's no assurance that

```

(Permission granted to reproduce and distribute as a complete document, without modification)

- \* handling of periodic events or aperiodic events will keep up with their fire() rates either.
- \* So I'm moving these services up to here.
- \*
  - \* ♦ The ReleaseParameters properties are considered to be final.
  - \* ♦ There will be no dynamic expansion of the arrival-time queue as the dynamic resource needs are unpredictable, and where are these resources going to come from, anyhow?
  - \* ♦ We will not support "exception" overflow behavior as distinct from replace or ignore. If you throw an exception, you still have to decide what to do with the overflow event, whether to ignore it or to ignore something else which it replaces. To be perfectly clear, we will allow the programmer to specify whether to ignore or replace in the case of an overflowing event, and we will throw an exception in either case. As with the RTSJ, if the event is triggered by a happening (or by an interrupt) and this event overflows the event triggering queue, no exception will be thrown, and we will either replace or ignore the new happening depending on the configuration of the release parameters.
  - \* ♦ Permit only the ignore and replace on overflow behaviors. This means we don't provide definitions for other options, and they won't be allowed.
- \* In summary, each event handler maintains its own pending event queue. When this queue
  - \* overflows, we do one of the following, as characterized by the configuration of this object:
  - \*
    - \* ♦ The new event is ignored if this handler is configured with arrivalTimeQueueOverflowIgnore.
    - \* ♦ The new event overwrites the oldest pending event if this handler is configured with arrivalTimeQueueOverflowReplace.
- \* Regardless of whether we ignore or replace the overflow event, the corresponding
  - \* AsyncEvent.fire() method always throws the overflow exception. It's not clear from
  - \* reading the RTSJ spec that this is the expected behavior.
- \* Also unclear from the RTSJ spec is when exactly the deadline miss handler is invoked. We
  - \* adopt the following conventions for the safety- and mission-critical profiles:
  - \*
    - \* ♦ We use javax.realtime.util.sc.Clock.getTimeKeeper() to measure deadline compliance.
    - \* ♦ In terms of this clock's tick intervals, the first tick counts as time zero. The second tick counts as one tick interval, and so on.
    - \* ♦ Deadline overrun is checked conservatively, meaning that we will not signal a deadline overrun unless we are absolutely sure that the deadline has been violated. This means that many missed deadlines will go undetected. This is an inherent limitation of modern implementation techniques for real-time software. At least, we will be clear about the semantics that are being implemented.
    - \* ♦ Our general algorithm is to take the desired relative deadline, divide this by the clock's tick interval, rounding up to the nearest integer, and add 1 tick. We consider the deadline to have been violated if this many ticks occur before the task has completed its execution.
    - \* ♦ There is no proactive enforcement of deadline compliance. Deadline compliance is not

```

*   checked until after the real-time task has completed its execution. At that time, we ask
*   whether the task completed on schedule. If not, we fire the miss_handler at this time.
*   This means that a real-time task that goes into an infinite loop or that deadlocks will
*   never have its miss handler fired. It also means we do not incur the burden of launching
*   a watchdog timer to monitor each and every execution of a real-time task, as this would
*   have a significant negative impact on overall system throughput. Programmers who
*   desire parallel watchdog monitoring of their real-time tasks can explicitly program this
*   by firing multiple event handlers off of the same event.
*/

```

```
public class ReleaseParameters extends javax.realtime.ReleaseParameters {
```

```

    public static final String arrivalTimeQueueOverflowIgnore = "ignore";
    public static final String arrivalTimeQueueOverflowReplace = "replace";

```

```
/**
```

```

* Create a new ReleaseParameters with the given values. The queue overflow behavior is set
* to the default value of arrivalTimeQueueOverflowIgnore, and the queue length is set to
* one.

```

```
*
```

```
* Parameters:
```

```
*
```

```

* deadline: The latest permissible completion time measured from the release time of the
* associated invocation of the schedulable object. Changing the deadline might not
* take effect until after the expiration of the current deadline. If null, the default
* value is new instance of RelativeTime(period).

```

```
*
```

```

* miss_handler: This handler is invoked if the run() method of the schedulable object is still
* executing after the deadline has passed. Although minimum implementations do not
* consider deadlines in feasibility calculations, they must recognize variable deadlines
* and invoke the miss handler as appropriate. If null, no application event handler is
* executed on the miss deadline condition.

```

```
*
```

```

* throws java.lang.IllegalArgumentException if the time of deadline is less than or equal to
* zero.

```

```
*/
```

```
protected @StaticAnalyzable @ScopedPure
```

```

    ReleaseParameters(RelativeTime deadline, AsyncEventHandler miss_handler)
    throws IllegalArgumentException;

```

```
/**
```

```

* Create a new ReleaseParameters with the given values.

```

```
*
```

```
*
```

```
* Parameters:
```

```
*
```

```

* deadline: The latest permissible completion time measured from the release time of the
* associated invocation of the schedulable object. Changing the deadline might not
* take effect until after the expiration of the current deadline. If null, the default
* value is new instance of RelativeTime(period).

```

```
*
```

```

* miss_handler: This handler is invoked if the run() method of the schedulable object is still
* executing after the deadline has passed. Although minimum implementations do not
* consider deadlines in feasibility calculations, they must recognize variable deadlines

```

```

*      and invoke the miss handler as appropriate. If null, no application event handler is
*      executed on the miss deadline condition.
*
* arrival_time_queue_length: Each AsyncEventHandler keeps track of the time at which each
*      pending triggered event was fired. It uses this information to detect deadline
*      overruns. This parameter specifies the length of this queue. The queue must have at
*      least one entry.
*
* arrival_time_queue_overflow_behavior: When the arrival-time queue overflows, the
*      behavior can be configured to either ignore the new event, or to replace the oldest
*      pending event with the new event. These two responses are selected by passing as
*      this argument either the value of arrivalTimeQueueOverflowIgnore or
*      arrivalTimeQueueOverflowReplace respectively.
*
* throws java.lang.IllegalArgumentException Thrown if
*      deadline is less than or equal to zero
*
* throws java.lang.IllegalArgumentException if the time of deadline is less than or equal to
*      zero, or if the arrival_time_queue_length argument is less than one, or if the value
*      of arrival_time_queue_overflow_behavior does not equal either
*      arrivalTimeQueueOverflowIgnore or arrivalTimeQueueOverflowReplace.
*/
public @StaticAnalyzable @ScopedPure ReleaseParameters(RelativeTime deadline,
    AsyncEventHandler miss_handler, int arrival_time_queue_length,
    String arrival_time_queue_overflow_behavior) throws IllegalArgumentException;

/**
 * Gets a reference to the arrival_time_queue_overflow_behavior configuration.
 *
 * returns a reference to the value of arrival_time_queue_overflow_behavior.
 */
public @StaticAnalyzable @ScopedThis @Scoped
    String getArrivalTimeQueueOverflowBehavior();

/**
 * Gets the initial (and final, because this state variable cannot change) number of entries that
 * the arrival time queue can hold.
 *
 * returns The length of the arrival-time queue.
 */
public @StaticAnalyzable @ScopedThis int getInitialArrivalTimeQueueLength();
}

```

---

## Scoped.java

---

```

package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies parameters, local variables, instance variables, and method return
 * values that agree to follow the rules required to hold references to stack allocated objects.

```

```
* In particular, the contents of these variables will never be copied into variables that have
* not agreed to follow these same restrictive rules.
```

```
*/
```

```
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
  @Target({ElementType.PARAMETER, ElementType.LOCAL_VARIABLE,
           ElementType.METHOD, ElementType.FIELD}) public @interface Scoped {
}
```

---

---

### ScopedArray.java

---

```
package javax.realtime.util.sc;
```

```
import java.lang.annotation.*;
```

```
/**
```

```
* This annotation identifies parameters, local variables, instance variables, and return values
* that agree to follow the rules required to hold references to scope-allocated arrays which
* themselves contain references to scope-allocated objects. In particular, the contents of the
* reference array elements will never be copied into variables that have not agreed to follow
* these same restrictive rules, and the array reference itself will only be copied to other
* variables that are declared with the @ScopedArray attribute.
```

```
*/
```

```
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
  @Target({ElementType.PARAMETER, ElementType.LOCAL_VARIABLE,
           ElementType.METHOD, ElementType.FIELD}) public @interface ScopedArray {
}
```

---

---

### ScopedArrayLocal.java

---

```
package javax.realtime.util.sc;
```

```
import java.lang.annotation.*;
```

```
/**
```

```
* This annotation identifies parameters to @CallerAllocatedResult and
* @CallerAllocatedArrayResult methods and to @ScopedThis constructors that
* agree to follow the rules required to hold references to stack allocated
* objects only in local variables, never copying their values into any instance
* or static fields. The contents of these variables will never
* be copied into variables that have not agreed to follow these same
* restrictive rules.
```

```
*/
```

```
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
  @Target({ElementType.PARAMETER}) public @interface ScopedArrayLocal {
}
```

---

---

## ScopedLocal.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies parameters to @CallerAllocatedResult and
 * @CallerAllocatedArrayResult methods and to @ScopedThis constructors that
 * agree to follow the rules required to hold references to stack allocated
 * objects only in local variables, never copying their values into any instance
 * or static fields. The contents of these variables will never
 * be copied into variables that have not agreed to follow these same
 * restrictive rules.
 */
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
@Target({ElementType.PARAMETER}) public @interface ScopedLocal {

}
```

---

---

## ScopedMemorySize.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;
import javax.realtime.util.sc.StaticLimit;

/**
 * Annotate a method or constructor with this annotation to specify the desired
 * size of the memory scope associated with the invocation.
 */
@Documented
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME) @Inherited public @interface ScopedMemorySize {

    /**
     * Reserve space for bytes() bytes of memory
     */
    public int bytes() default 0;

    /**
     * Let N be the length of array instances(). Reserve space for instances[i]
     * occurrences of types[i], for i ranging from 0 to N-1
     */
    public int[] instances() default {};
    public Class[] types() default {};

    /**
     * Let N be the length of reference_array_instances(). Reserve the amount of
     * memory represented by summing the total memory required to represent
     * reference_array_instances[i] reference arrays of size
     */
}
```

```

    * reference_array_dimensions[i] for i ranging from 0 to N-1.
    */
    public int[] reference_array_instances() default {};
    public int[] reference_array_dimensions() default {};

    /**
     * Let N be the length of primitive_array_instances. Reserve the amount of
     * memory represented by summing the total memory required to represent
     * primitive_array_instances[i] primitive arrays of size
     * primitive_array_dimensions[i], with each array element of type
     * primitive_array_types[i], for i ranging from 0 to N-1.
     */
    public int[] primitive_array_instances() default {};
    public int[] primitive_array_dimensions() default {};
    public Class[] primitive_array_types() default {};

    /**
     * Let N be the length of method_invocations. Reserve the amount of stack memory
     * required to support a total of method_invocations[i] calls of the
     * method identified by method_signatures[i], for i ranging from 0 to N-1. Each
     * method is represented by its name alone if the name is unambiguous, and by
     * the name concatenated with a verbose description of the method's
     * parameterization if there are multiple methods by the same name. The
     * verbose parameterization simply lists the argument types using fully
     * qualified class names as a comma-selected list enclosed within parenthesis.
     * For example: foo(int,java.lang.String,int[])
     */
    public int[] method_invocations() default {};
    public String[] method_signatures() default {};
}

```

---

## ScopedPure.java

---

```

package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * When present, this annotation denotes that all arguments, including this for constructors and
 * virtual methods, follow the rules required to hold references to stack allocated objects. In
 * particular, the contents of these variables will never be copied into variables that have not
 * agreed to follow these same restrictive rules.
 *
 * This annotation does not necessarily denote that return results are stack allocated. If the
 * desire is to stack-allocate the return result, that must be denoted with a separate
 * @CallerAllocatedResult annotation.
 */
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface ScopedPure {
}

```

---

---

## ScopedThis.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies virtual methods that agree to abide by the rules required to safely
 * allow its implicit this variable to refer to a stack-allocated object.
 */
@Documented @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.CLASS) @Inherited public @interface ScopedThis {
}
```

---

---

## ScopedThisLocal.java

---

```
package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * This annotation identifies @CallerAllocatedResult and
 * @CallerAllocatedArrayResult methods that agree to follow the rules required
 * to hold the value of this as a reference to a stack allocated object,
 * refraining from copying this value into any instance or static fields. The
 * contents of this will never be copied to any variable that has not agreed to
 * follow these same restrictive rules.
 */
@Documented @Retention(RetentionPolicy.CLASS) @Inherited
@Target({ElementType.PARAMETER}) public @interface ScopedThisLocal {
}
```

---

---

## SizeEstimator.java

---

```
package javax.realtime.util.sc;

/**
 * This class mimics the behavior of javax.realtime.SizeEstimator.
 *
 * * RTSJ additions:
 * * ♦ NO_BOUND
 * * ♦ ZERO
 * * ♦ SizeEstimator reserveBytes(int number)
 * * ♦ reserveStack(aeh)
 * * ♦ reserveStack(runcode)
 *
 * * RTSJ refinements:
 * * ♦ There's a misleading/confusing comment in the RTSJ specification about these values

```

- \* representing a "floor on the amount of memory that should be allocated". Their point is
- \* that creation of one object may actually require creation of other objects to which the
- \* first object will refer. That point is well taken, but it simply needs to be clarified to the
- \* developer that he is responsible for discovering all of the objects that need to be
- \* created, both directly and indirectly, and accumulating all of these objects into the
- \* SizeEstimator tally.
- \*
- \* ♦ The RTSJ also states that "alignment considerations, and possibly other order-
- \* dependent issues [e.g. fragmentation?] may cause the allocator to leave a small amount of
- \* unusable space. Consequently, the size estimate cannot be seen as more than a close
- \* estimate." This doublespeak leaves too much imprecision in the specified behavior of
- \* SizeEstimator. We shall require that SizeEstimator include in its analysis of each
- \* object's size whatever implementation-dependent alignment padding and/or bookkeeping
- \* overhead might, in the worst case, be affiliated with the allocation of each object. This
- \* means the SizeEstimator's analysis always represents a ceiling on the total amount of
- \* memory that must be allocated. Fragmentation effects are not captured in
- \* the SizeEstimator analysis. But those can be handled separately, by using LIFO (stack-
- \* based) allocation, for example.
- \*
- \* ♦ It bothers me that SizeEstimator is not used consistently throughout the RTSJ API.
- \* There are many methods that speak of memory sizes in terms of long integer byte
- \* counts. In all such cases, there ought to be comparable methods that speak in terms of
- \* SizeEstimator arguments.
- \*
- \* ♦ Maybe it doesn't matter, but it sure seems these methods could possibly result in
- \* ArithmeticException, at least if you've got some sloppy miscalculations in your memory
- \* requirements approximation.
- \*

```

*/
public class SizeEstimator extends javax.realtime.SizeEstimator {

    /**
     * A symbolic constant representing that the memory usage is unbounded.
     */
    public final static SizeEstimator NO_BOUND;;

    /**
     * A symbolic constant representing that no memory allocation is required.
     */
    public final static SizeEstimator ZERO;

    /**
     * Create a SizeEstimator object.
     */
    public @StaticAnalyzable @ScopedThis SizeEstimator();

    /**
     * Take into account an additional number of bytes of memory in the total amount of memory
     * represented by this object.
     *
     * Parameters:
     *
     * number: The number of bytes to add in to the total.
     */

```

```

public @StaticAnalyzable @ScopedThis void reserveBytes(int number);

/**
 * In a statically analyzed system, all scope sizes are determined by static
 * analysis of software. However, there are certain situations in which scope
 * sizes cannot be determined until run time. Use this service to adjust scope
 * sizes at run time.
 *
 * This method allows program components to dynamically set the size of the
 * AllocationContext. Only one occurrence of ensureScopeCapacity() is
 * permitted within each constructor of a (@ReentrantScope) object and within
 * each method of a non-@ReentrantScope object. ensureScopeCapacity() may not
 * be invoked from any other context. It is also disallowed within any method
 * that includes the (@StaticAnalyzable) or (@ScopedMemorySize) annotations.
 * The invocation of ensureScopeCapacity() must dominate all allocations
 * within the scope except for optional instantiation of a single (@Scoped)
 * SizeEstimator object.
 *
 * Returns true if the size is at least as large as is specified by the s
 * argument. Returns false if there is not sufficient memory to ensure the
 * requested capacity.
 *
 * s: The required size of the current ScopedMemory region
 *
 * returns true iff if the current ScopedMemory region is at least as large as indicated by s
 */
public static boolean ensureScopeCapacity(SizeEstimator s);
}

```

---

## SporadicParameters.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

/**
 * This class mimics javax.realtime.SporadicParameters.
 *
 * RTSJ additions:
 *
 * ♦ SporadicParameters(min_interarrival, deadline, miss_handler,
 *     mit_violation_handler, queue_length, queue_overflow_behavior);
 *
 * RTSJ refinements:
 *
 * ♦ Note that arrival-time queue overflow handling has been "hoisted" into the
 *   ReleaseParameters definition
 *
 * ♦ We don't provide any options for configuring detected violations of mit. Instead, we'll
 *   expect the semantics described for mitViolationIgnore, with a special
 *   mit_violation_handler fired each time this occurs.

```

(Permission granted to reproduce and distribute as a complete document, without modification)

```
*/
public class SporadicParameters extends javax.realtime.SporadicParameters {
/**
 * Create a SporadicParameters object. The queue overflow behavior is set to the default
 * value of arrivalTimeQueueOverflowIgnore and the queue length is set to one.
 *
 * Parameters:
 *
 * min_interarrival: The release times of the schedulable object will occur no closer than this
 * interval. This time object is treated as if it were copied. The constructed object
 * does not retain a reference to it.
 *
 * deadline: The latest permissible completion time measured from the release time of the
 * associated invocation of the schedulable object. If null, the default value is a new
 * instance of RelativeTime(Long.MAX_VALUE, 999999).
 *
 * miss_handler This handler is invoked if the run() method of the schedulable object is still
 * executing after the deadline has passed. If null, no deadline miss handler will be
 * invoked, and deadline compliance checking is not implemented.
 *
 * mit_violation_handler: This handler is invoked each time consecutive triggerings of the
 * corresponding event are closer to each other than the min_interarrival time. If null,
 * no handler will be invoked in this circumstance.
 *
 * throws java.lang.IllegalArgumentException if min_interarrival or deadline is less than or
 * equal to zero.
 */
public @StaticAnalyzable @ScopedPure
    SporadicParameters(RelativeTime min_interarrival, RelativeTime deadline,
        AsyncEventHandler miss_handler, AsyncEventHandler mit_violation_handler);

/**
 * Create a SporadicParameters object.
 *
 * Parameters:
 *
 * min_interarrival: The release times of the schedulable object will occur no closer than this
 * interval. This time object is treated as if it were copied. The constructed object
 * does not retain a reference to it.
 *
 * deadline: The latest permissible completion time measured from the release time of the
 * associated invocation of the schedulable object. If null, the default value is a new
 * instance of RelativeTime(Long.MAX_VALUE, 999999).
 *
 * miss_handler: This handler is invoked if the run() method of the schedulable object is still
 * executing after the deadline has passed. If null, no deadline miss handler will be
 * invoked, and deadline compliance checking is not implemented.
 *
 * mit_violation_handler: This handler is invoked each time consecutive triggerings of the
 * corresponding event are closer to each other than the min_interarrival time. If null,
 * no handler will be invoked in this circumstance.

```

```

*
* arrival_time_queue_length: Each AsyncEventHandler keeps track of the time at which each
*   pending triggered event was fired. It uses this information to detect deadline
*   overruns. This parameter specifies the length of this queue. The queue must have at
*   least one entry.
*
* queue_overflow_behavior: When the arrival-time queue overflows, the behavior can be
*   configured to either ignore the new event, or to replace the oldest pending event
*   with the new event. These two responses are selected by passing as this argument
*   either the value of ReleaseParameters.arrivalTimeQueueOverflowIgnore or
*   ReleaseParameters.arrivalTimeQueueOverflowReplace respectively.
*
* throws java.lang.IllegalArgumentException if min_interarrival or deadline is less than or
*   equal to zero, or if the arrival_time_queue_length argument is less than one, or if
*   the value of arrival_time_queue_overflow_behavior does not equal one of
*   ReleaseParameters.arrivalTimeQueueOverflowIgnore or
*   ReleaseParameters.arrivalTimeQueueOverflowReplace.
*/
public @StaticAnalyzable @ScopedPure
    SporadicParameters(RelativeTime min_interarrival, RelativeTime deadline,
        AsyncEventHandler miss_handler, AsyncEventHandler mit_violation_handler,
        int arrival_time_queue_length, String queue_overflow_behavior);

/**
 * Gets a reference to the arrival_time_queue_overflow_behavior configuration.
 *
 * returns a reference to the value of arrival_time_queue_overflow_behavior.
 */
public @StaticAnalyzable @ScopedThis String getArrivalTimeQueueOverflowBehavior();

/**
 * Gets the initial (and final, because this state variable cannot change) number of entries that
 * the arrival time queue can hold.
 *
 * returns the length of the arrival-time queue.
 */
public @StaticAnalyzable @ScopedThis int getInitialArrivalTimeQueueLength();
}

```

---

## StaticAnalyzable.java

---

```

package javax.realtime.util.sc;

import java.lang.annotation.*;
import javax.realtime.util.sc.StaticLimit;

/**
 * Annotate a method or constructor with this annotation in order to request
 * that certain static properties of the method or constructor be automatically
 * determined by development tools.
 */

```

```
@Documented
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME) @Inherited public @interface StaticAnalyzable {

    /**
     * The length of this array must be either 1 or must equal the number of entries in the
     * modes enumeration. Otherwise, the byte-code verifier will reject this declaration.
     * If the length of this array is 1, but the enumeration has more than entry, the value of
     * the single entry corresponds to all analysis modes.
     *
     * If the value of this attribute is true for a particular analysis mode, the byte code verifier
     * rejects as illegal any method for which its analysis cannot derive upper bounds on execution
     * time.
     */
    boolean[] enforce_time_analysis() default {true};

    /**
     * The length of this array must be either 1 or must equal the number of entries in the
     * modes enumeration. Otherwise, the byte-code verifier will reject this declaration.
     * If the length of this array is 1, but the enumeration has more than entry, the value of
     * the single entry corresponds to all analysis modes.
     *
     * If this code can be proven to be non-blocking, then it can be safely
     * invoked from within a context that holds an Atomic priority ceiling lock.
     * Otherwise, it should not be invoked from within such a context because
     * blocking would compromise the desired efficient PCP implementation.
     *
     * TRUE means we enforce non-blocking behavior. FALSE means we do not
     * enforce non-blocking behavior. FALSE does not necessarily mean
     * that we prove blocking behavior.
     */
    boolean[] enforce_non_blocking() default {true};

    /**
     * The length of this array must be either 1 or must equal the number of entries in the
     * modes enumeration. Otherwise, the byte-code verifier will reject this declaration.
     * If the length of this array is 1, but the enumeration has more than entry, the value of
     * the single entry corresponds to all analysis modes.
     *
     * If this attribute is true, the byte code verifier rejects as illegal any
     * method for which its analysis cannot derive upper bounds on stack growth or
     * immortal memory consumption.
     */
    boolean[] enforce_memory_analysis() default {true};

    /**
     * Developers may estimate CPU time requirements and provide this information
     * for use by other static analysis tools by initializing this attribute. This
     * practice is strongly discouraged because (1) developers may error in their
     * analysis and (2) the results of their analysis are very non-portable.
     */
    long[] user_estimated_time() default {StaticLimit.UNANALYZED_TIME};
}
```

```
/**
 * Developers may estimate ImmortalMemory requirements and provide this information
 * for use by other static analysis tools by initializing this attribute. This
 * practice is strongly discouraged because (1) developers may error in their
 * analysis and (2) the results of their analysis are very non-portable.
 */
long[] user_estimated_heap() default {StaticLimit.UNANALYZED_SIZE_BYTES};
```

```
/**
 * Developers may estimate stack memory requirements and provide this information
 * for use by other static analysis tools by initializing this attribute. This
 * practice is strongly discouraged because (1) developers may error in their
 * analysis and (2) the results of their analysis are very non-portable.
 */
long[] user_estimated_stack() default {StaticLimit.UNANALYZED_SIZE_BYTES};
```

```
/**
 * When declaring a method or constructor to be analyzable, the developer may
 * identify multiple distinct characteristic behaviors. The results of analysis are likely
 * to depend on the current system state and/or parameter
 * values. The modes argument provides enumeration constants to
 * represent each of the distinct modes of analysis.
 *
 * The first element of the enumeration always represents the default mode,
 * which generally, but does not necessarily, represent the worst possible
 * resource requirements.
 */
java.lang.Class< ? extends java.lang.Enum> modes()
    default StaticLimit.DefaultAnalysisMode.class;
```

```
/**
 * Represents the nanoseconds required to execute this method's code, in each
 * of the identified modes of operation.
 *
 * See StaticLimit.UNANALYZED_TIME
 * See StaticLimit.UNANALYZABLE_TIME
 */
long[] execution_time() default {};
```

```
/**
 * Represents the maximum number of bytes of stack memory required to execute
 * this method, including all methods that are invoked from this method.
 *
 * See StaticLimit.UNANALYZED_SIZE_BYTES
 * See StaticLimit.UNANALYZABLE_SIZE_BYTES
 */
long[] stack_bytes() default {};
```

```

/**
 * Represents the maximum number of bytes of "heap" memory required to execute
 * this method, including all methods that are invoked from this method. We
 * use the term "heap" loosely. This heap does not necessarily refer to Java's
 * garbage collected heap. Rather, it refers to the ImmortalMemory region.
 *
 * See StaticLimit.UNANALYZED_SIZE_BYTES
 * See StaticLimit.UNANALYZABLE_SIZE_BYTES
 */
long[] heap_bytes() default {};
}

```

---



---

### StaticDependency.java

---

```

package javax.realtime.util.sc;

import java.lang.annotation.*;

/**
 * Annotate a static variable to indicate to the static linker that certain other static variables,
 * as identified by the c and field attributes, must be initialized before this static variable is
 * initialized.
 */
@Target({ElementType.FIELD}) @Retention(RetentionPolicy.CLASS) @Inherited @Documented
public @interface StaticDependency {

    java.lang.Class c();

    java.lang.String field();

}

```

---



---

### StaticLimit.java

---

```

package javax.realtime.util.sc;

/**
 * This class is used to write assertions relating to analysis of static properties in real-time Java
 * code. Among the static properties that can be automatically derived are:
 *
 * ♦ The maximum amount of CPU time required to execute the method
 *
 * ♦ The maximum amount of memory allocated on the run-time stack during execution of this
 * method, and
 *
 * ♦ The maximum amount of memory allocated in the heap during execution of this method.
 */

```

- \* For certain of these static properties, the worst-case resource requirements will depend on
- \* aspects of the input data or system state. Programmers may distinguish between multiple
- \* modes of operation for each method by providing a custom enumeration in the
- \* parameterization of the method's @StaticAnalyzable annotation.

- \* Many of the StaticLimit methods include an Enum mode argument which has the effect of
- \* making the assertion contingent upon a particular analysis mode.

```
public class StaticLimit {

    public enum DefaultAnalysisMode { CONSERVATIVE };

    /**
     * UNANALYZABLE_TIME means the corresponding program component does
     * not follow the guidelines that are required to support automatic
     * analysis of worst-case execution time.
     */
    public static final long UNANALYZABLE_TIME = -1L;

    /**
     * UNANALYZED_TIME means the corresponding program component does
     * follow the guidelines that are required to support automatic
     * analysis of worst-case execution time, and thus the worst-case
     * execution time is analyzable. However, due to limits of the
     * development environment and/or run-time environment, this program
     * component has not been analyzed.
     */
    public static final long UNANALYZED_TIME;

    /**
     * UNANALYZABLE_SIZE_BYTES means the corresponding program component does not
     * follow the guidelines that are required to support automatic analysis of worst-case memory
     * allocation needs.
     */
    public static final long UNANALYZABLE_SIZE_BYTES;

    /**
     * UNANALYZED_SIZE_BYTES means the corresponding program component does follow the
     * guidelines that are required to support automatic analysis of worst-case memory allocation
     * needs, and thus the worst-case memory allocation needs are analyzable. However, due
     * to limits of the development environment and/or run-time environment, this program
     * component has not been analyzed.
     */
    public static final long UNANALYZED_SIZE_BYTES ;

    /**
     * For the invocation that immediately follows this assertion, assume that the invoked
     * method's static properties when invoked from this particular context are best represented
     * by analyzing the method in analysis mode callee_mode.
     */
    public static boolean InvocationMode(@Scoped Enum callee_mode);

    /**
```

```
* For the invocation that immediately follows this assertion, when this method is being
* analyzed in mode caller_mode, assume that the invoked method's static properties when
* invoked from this particular context are best represented by analyzing that method
* according to analysis mode callee_mode.
*/
public @ScopedPure static boolean InvocationMode(Enum caller_mode, Enum callee_mode);

/**
 * For each time the inner-most enclosing loop is entered, this particular assertion is executed
 * no more than max_iterations times. Note that execution of this assertion may be
 * conditionally executed (as part of an if-then-else statement, for example. In this case, the
 * max_iterations parameter specifies a bound on the number of times the controlling
 * condition is satisfied rather than limiting the loop itself.
 */
public static boolean IterationBound(int max_iterations);

/**
 * For each time the inner-most enclosing loop is entered, this particular assertion is executed
 * no more than max_iterations times when this method is running in the mode specified by
 * the mode parameter. Note that execution of this assertion may be conditionally executed
 * (as part of an if-then-else statement, for example. In this case, the max_iterations
 * parameter specifies a bound on the number of times the controlling condition is satisfied
 * rather than limiting the loop itself.
 */
public @ScopedPure static boolean IterationBound(Enum mode, int max_iterations);

/**
 * For each time the loop at nesting level nesting_level relative to the context of this
 * assertion is entered, this particular assertion is executed no more than max_iterations
 * times. nesting_level zero identifies the inner-most nesting level. Note that execution of
 * this assertion may be conditionally executed (as part of an if-then-else statement, for
 * example. In this case, the max_iterations parameter specifies a bound on the number of
 * times the controlling condition is satisfied rather than limiting the loop itself.
 */
public static boolean NestedIterationBound(int nesting_level, int max_iterations);

/**
 * For each time the loop at nesting level nesting_level relative to the context of this
 * assertion is entered, this particular assertion is executed no more than max_iterations
 * times when this method is executing in the mode specified by the mode parameter.
 * nesting_level zero identifies the inner-most nesting level. Note that execution of this
 * assertion may be conditionally executed (as part of an if-then-else statement, for example.
 * In this case, the max_iterations parameter specifies a bound on the number of times the
 * controlling condition is satisfied rather than limiting the loop itself.
 */
public @ScopedPure static
    boolean NestedIterationBound(Enum mode, int nesting_level, int max_iterations);

/**
 * This indicates that the basic block of code that contains this assertion will never execute.
 *
 * returns false, because if this code executes, that represents a violation of the indicated
 * programmer's intent.
 */
```

```
*/  
  
/**  
 * This indicates that the basic block of code that contains this assertion will not execute in  
 * the indicated analysis mode.  
 *  
 * Parameters:  
 *  
 * mode: The analysis mode under which this particular basic block should not be executed.  
 *  
 * returns false if the program's current mode of execution matches the value of the mode  
 * argument. Otherwise, returns true.  
 */  
public static boolean NotReached(@Scoped Enum mode);  
  
/**  
 * This assertion, normally placed immediately following the allocation of a new array of  
 * boolean, indicates that the array has no more than bound elements.  
 *  
 * Parameters:  
 *  
 * ba: The array whose size is to be limited.  
 *  
 * bound: The maximum number of elements in this array.  
 *  
 * returns true if the array ba has no more than bound elements. Otherwise, returns false.  
 */  
public static boolean ArrayLength(@Scoped boolean ba[], int bound);  
  
/**  
 * This assertion, normally placed immediately following the allocation of a new array of  
 * boolean, indicates that the array has no more than bound elements when this method is  
 * executing according to analysis mode mode.  
 *  
 * Parameters:  
 *  
 * mode: The analysis mode under which this particular assertion should be enforced.  
 *  
 * ba: The array whose size is to be limited.  
 *  
 * bound: The maximum number of elements in this array.  
 *  
 * returns true if the program's current mode of execution does not match the value of the  
 * mode argument or if the array ba has no more than bound elements. Otherwise,  
 * returns false.  
 */  
public @ScopedPure static boolean ArrayLength(Enum mode, boolean ba[], int bound);  
  
/**  
 * This assertion, normally placed immediately following the allocation of a new byte array,  
 * indicates that the array has no more than bound elements.  
 *  
 * Parameters:
```

```
*
* ba: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the array ba has no more than bound elements. Otherwise, returns false.
*/
public static boolean ArrayLength(@Scoped byte ba[], int bound);

/**
* This assertion, normally placed immediately following the allocation of a new byte array,
* indicates that the array has no more than bound elements when this method is executing
* according to analysis mode mode.
*
* Parameters:
*
* mode: The analysis mode under which this particular assertion should be enforced.
*
* ba: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the program's current mode of execution does not match the value of the
* mode argument or if the array ba has no more than bound elements. Otherwise,
* returns false.
*/
public @ScopedPure static boolean ArrayLength(Enum mode, byte ba[], int bound);

/**
* This assertion, normally placed immediately following the allocation of a new array of char,
* indicates that the array has no more than bound elements.
*
* Parameters:
*
* ca: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the array ca has no more than bound elements. Otherwise, returns false.
*/
public static boolean ArrayLength(@Scoped char ca[], int bound);

/**
* This assertion, normally placed immediately following the allocation of a new array of char,
* indicates that the array has no more than bound elements when this method is executing
* according to analysis mode mode.
*
* Parameters:
*
* mode: The analysis mode under which this particular assertion should be enforced.
*
* ca: The array whose size is to be limited.
*
```

```
* bound: The maximum number of elements in this array.
*
* returns true if the program's current mode of execution does not match the value of the
*     mode argument or if the array ca has no more than bound elements. Otherwise,
*     returns false.
*/
public static boolean ArrayLength(Enum mode, @Scoped char ca[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of short,
 * indicates that the array has no more than bound elements.
 *
 * Parameters:
 *
 * sa: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the array sa has no more than bound elements. Otherwise, returns false.
 */
public static boolean ArrayLength(@Scoped short sa[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of short,
 * indicates that the array has no more than bound elements when this method is executing
 * according to analysis mode mode.
 *
 * Parameters:
 *
 * mode: The analysis mode under which this particular assertion should be enforced.
 *
 * sa: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the program's current mode of execution does not match the value of the
 *     mode argument or if the array sa has no more than bound elements. Otherwise,
 *     returns false.
 */
public static boolean ArrayLength(Enum mode, @Scoped short sa[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of int,
 * indicates that the array has no more than bound elements.
 *
 * Parameters:
 *
 * ia: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the array ia has no more than bound elements. Otherwise, returns false.
 */
```

```
public static boolean ArrayLength(@Scoped int ia[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of int,
 * indicates that the array has no more than bound elements when this method is executing
 * according to analysis mode mode.
 *
 * Parameters:
 *
 * mode: The analysis mode under which this particular assertion should be enforced.
 *
 * ia: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the program's current mode of execution does not match the value of the
 * mode argument or if the array ia has no more than bound elements. Otherwise,
 * returns false.
 */
public static boolean ArrayLength(Enum mode, @Scoped int ia[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of float,
 * indicates that the array has no more than bound elements.
 *
 * Parameters:
 *
 * fa: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the array ba has no more than bound elements. Otherwise, returns false.
 */
public static boolean ArrayLength(@Scoped float fa[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of float,
 * indicates that the array has no more than bound elements when this method is executing
 * according to analysis mode mode.
 *
 * Parameters:
 *
 * mode: The analysis mode under which this particular assertion should be enforced.
 *
 * fa: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the program's current mode of execution does not match the value of the
 * mode argument or if the array fa has no more than bound elements. Otherwise,
 * returns false.
 */
public static boolean ArrayLength(Enum mode, @Scoped float fa[], int bound);
```

```
/**
 * This assertion, normally placed immediately following the allocation of a new array long,
 * indicates that the array has no more than bound elements.
 *
 * Parameters:
 *
 * la: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the array ba has no more than bound elements. Otherwise, returns false.
 */
public static boolean ArrayLength(@Scoped long la[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of long,
 * indicates that the array has no more than bound elements when this method is executing
 * according to analysis mode mode.
 *
 * Parameters:
 *
 * mode: The analysis mode under which this particular assertion should be enforced.
 *
 * la: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the program's current mode of execution does not match the value of the
 * mode argument or if the array la has no more than bound elements. Otherwise,
 * returns false.
 */
public static boolean ArrayLength(Enum mode, @Scoped long la[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of double,
 * indicates that the array has no more than bound elements.
 *
 * Parameters:
 *
 * da: The array whose size is to be limited.
 *
 * bound: The maximum number of elements in this array.
 *
 * returns true if the array da has no more than bound elements. Otherwise, returns false.
 */
public static boolean ArrayLength(@Scoped double da[], int bound);

/**
 * This assertion, normally placed immediately following the allocation of a new array of double,
 * indicates that the array has no more than bound elements when this method is executing
 * according to analysis mode mode.
 *
 *
 */
```

```

* Parameters:
*
* mode: The analysis mode under which this particular assertion should be enforced.
*
* da: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the program's current mode of execution does not match the value of the
*     mode argument or if the array da has no more than bound elements. Otherwise,
*     returns false.
*/
public static boolean ArrayLength(Enum mode, @Scoped double da[], int bound);

/**
* This assertion, normally placed immediately following the allocation of a new array of
* references, indicates that the array has no more than bound elements.
*
* Parameters:
*
* oa: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the array oa has no more than bound elements. Otherwise, returns false.
*/
public static boolean ArrayLength(@Scoped Object oa[], int bound);

/**
* This assertion, normally placed immediately following the allocation of a new array of
* references, indicates that the array has no more than bound elements when this method is
* executing according to analysis mode mode.
*
* Parameters:
*
* mode: The analysis mode under which this particular assertion should be enforced.
*
* oa: The array whose size is to be limited.
*
* bound: The maximum number of elements in this array.
*
* returns true if the program's current mode of execution does not match the value of the
*     mode argument or if the array oa has no more than bound elements. Otherwise,
*     returns false.
*/
public static boolean ArrayLength(Enum mode, @Scoped Object oa[], int bound);
}

```

---

## TimeBuffer.java

---

```
package javax.realtime.util.sc;
```

```
/**
 * This class has no analog in the RTSJ. It is provided as a notational convenience to encourage
 * idiomatic usage with respect to creation of RelativeTime values.
 */
public class TimeBuffer {

    /**
     * Construct a new TimeBuffer object
     */
    public @StaticAnalyzable @ScopedThis TimeBuffer();

    /**
     * Construct a new TimeBuffer object with the specified initial
     * value.
     *
     * Parameters:
     *
     * millis: The number of milliseconds in the initial value.
     *
     * nanos: The number of nanoseconds to be added to the milliseconds in the initial value.
     */
    public @StaticAnalyzable @ScopedThis TimeBuffer(long millis, int nanos);

    /** Add d days to this TimeBuffer object.
     *
     * Parameters:
     *
     * d: The number of days to be added to the time buffer.
     *
     * returns the modified TimeBuffer object.
     */
    public final @StaticAnalyzable @ScopedThis TimeBuffer d(int d);

    /** Add h hours to this TimeBuffer object.
     *
     * Parameters:
     *
     * h: The number of hours to be added to the time buffer.
     *
     * returns the modified TimeBuffer object.
     */
    public final @StaticAnalyzable @ScopedThis TimeBuffer h(int h);

    /**
     * Add the amount of time equal to one period of a periodic activity running at the specified
     * Hertz rate.
     *
     * Parameters:
     *
     * hertz: The cycle rate per second, from which the period is calculated.
     *
     * returns the modified TimeBuffer object.
     */
}
```

```
*/
public final @StaticAnalyzable @ScopedThis TimeBuffer hertz(int hertz);

/**
 * Add m minutes to this TimeBuffer object.
 *
 * Parameters:
 *
 * m: The number of minutes to add to the time buffer.
 *
 * returns the modified TimeBuffer object.
 */
public final @StaticAnalyzable @ScopedThis TimeBuffer m(int m);

/**
 * Add s seconds to this TimeBuffer object.
 *
 * Parameters:
 *
 * s: The number of seconds to add to the time buffer.
 *
 * returns the modified TimeBuffer object.
 */
public final @StaticAnalyzable @ScopedThis TimeBuffer s(int s);

/**
 * Add ms milliseconds to this TimeBuffer object.
 *
 * Parameters:
 *
 * ms: The number of milliseconds to add to the time buffer.
 *
 * returns the modified TimeBuffer object.
 */
public final @StaticAnalyzable @ScopedThis TimeBuffer ms(int ms);

/**
 * Add us microseconds to this TimeBuffer object.
 *
 * Parameters:
 *
 * us: The number of microseconds to add to the time buffer.
 *
 * returns the modified TimeBuffer object.
 */
public final @StaticAnalyzable @ScopedThis TimeBuffer us(int us);

/**
 * Add ns nanoseconds to this TimeBuffer object.
 *
 * Parameters:
 *
 * ns: The number of nanoseconds to add to the time buffer.
```

```

*
* returns the modified TimeBuffer object
*/
public final @StaticAnalyzable @ScopedThis TimeBuffer ns(int ns);

/**
 * Returns a string representation of the TimeBuffer, formatted according to the template:
 * "dddd hh:mm:ss.decimal", where:
 *   d represents days,
 *   h represents hours,
 *   m represents minutes,
 *   s represents seconds, and
 *   decimal represents the fractional number of seconds
 */
public final @StaticAnalyzable @CallerAllocatedResult @ScopedThis String toString();

/**
 * Copy the conversion of this TimeBuffer object into a RelativeTime object.
 *
 * Parameters:
 *
 * clock: The newly allocated RelativeTiem object will be associated with this clock.
 *
 * returns newly allocated RelativeTime object.
 */
public final @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    RelativeTime relative(Clock clock);

/**
 * Copy the conversion of this TimeBuffer object into an
 * AbsoluteTime object.
 *
 * Parameters:
 *
 * clock: The newly allocated RelativeTiem object will be associated with this clock.
 *
 * returns a newly allocated AbsoluteTime object representing the specified Clock's notion of
 *   current time plus the offset represented by this TimeBuffer.
 */
public final @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    AbsoluteTime absolute(Clock clock);
}

```

---



---

## Timer.java

---

```

package javax.realtime.util.sc;

import javax.realtime.AsyncEventHandler;

/**
 * This class mimics javax.realtime.Timer.
 * <p>

```

\* RTSJ additions:

```
* ♦ Timer(HighResolutionTime, AsyncEventHandler);
* ♦ Timer(HighResolutionTime, AsyncEventHandler []);
* ♦ getPreviousFireTime();
* ♦ getUserFireTime();
*
```

\* RTSJ refinements:

```
*
* ♦ In a real system, it is most likely that at least one clock is read-only, accumulating time
* with a hardware counter rather than software interrupt ticks. Such a clock cannot be
* used to drive timeout events. Nevertheless, we want to represent that clock with a Clock
* object so that we can query the high-precision time. The relevance here is that
* construction of a Timer object will fail if you attempt to use a non-ticking hardware
* clock as an argument to the constructor.
*
* ♦ Clarify semantics of reschedule(). Is this "atomic" in the sense that:
*
*   ♦ If my new time is later than my current time, I'm guaranteed not to "miss" the event
*     firing, and
*   ♦ If my new time is expressed as a RelativeTime, I'm guaranteed not to "miss" the
*     event firing.
*
* ♦ This second condition should also be clarified for the initial start condition. Another
* question to clarify: what is the time complexity of reschedule()? Is the work involved
* equivalent to canceling and restarting the timer?
*
* ♦ Another point of clarification: I wonder if the RTSJ description of isRunning() is
* sufficiently precise. In one description, they say "Is firing expected?" In another, they
* say "if this is active and is enabled". Treating these two statements as identical assumes
* a particular execution model which is not specified by the RTSJ. What if the Timer is
* enabled and active but the event time has already passed, for example?
*
* ♦ To support scalable and portable development, it is essential to clarify the resource
* requirements of Timer implementations. I propose to require complying implementations
* to support the following behaviors:
*
*   ♦ The CPU time to instantiate a new timer, or to start() a timer that has been
*     deactivated is proportional to the logarithm of the number of timers already
*     associated with this clock.
*
*   ♦ The CPU time to enable() or disable() an existing timer, and the CPU time to stop() or
*     destroy() a timer is constant.
*
*   ♦ The CPU time consumed at each tick is proportional to the number of Timers that
*     must be fired on that tick. (It is not proportional to the total number of active
*     Timers associated with this clock.)
*/
```

```
public class Timer extends javax.realtime.Timer {
/**
```

```
* Create a new Timer object, to execute its own fire() method at the time specified by
* argument t, and triggering execution of handler each time this event is fired. This timer
* will be associated with the same clock that is affiliated with its argument t.
*
```

```

* The time complexity of this operation needs to be proportional to the logarithm of the
* number of previously scheduled timer events associated with this clock that are still
* pending.
*
* Parameters:
*
* t: The time at which this Timer will execute its own fire() method.
*
* handler: The code to be triggered each time this object executes its fire() method.
*
* throws IllegalArgumentException if the clock associated with argument t is not an active
*         clock (i.e. if it does not generate timer ticks).
*/
public @ScopedPure Timer(HighResolutionTime t, AsyncEventHandler handler)
    throws IllegalArgumentException;

/**
* Create a new Timer object, to execute its own fire() method at the time specified by
* argument t, and triggering execution of each event handler in the handlers array each time
* this event is fired. This timer will be associated with the same clock that is affiliated with
* its argument t.
*
* Parameters:
*
* t: The time at which this Timer will execute its own fire() method.
*
* handlers: Represents the array of event handlers, all of which will be triggered for
*             execution each time this object executes its fire() method.
*
* throws IllegalArgumentException if the clock associated with argument t is not an active
*         clock (i.e. if it does not generate timer ticks).
*/
public @ScopedPure
    Timer(HighResolutionTime t, @ScopedArray AsyncEventHandler handlers[])
        throws IllegalArgumentException;

/**
* Returns the absolute past time at which this event last "fired".
*
* returns the AbsoluteTime representing the previous firing time of this Timer
*
* throws IllegalStateExpression Thrown if this event has not previously been fired.
*/
public @StaticAnalyzable @CallerAllocatedResult @ScopedThis
    AbsoluteTime getPreviousFireTime() throws IllegalStateException;

/**
* Returns a copy of the most recently requested fire time, in the same terms as were
* requested by the user of this Timer service (e.g. if the user specified a RelativeTime, this
* method will return a RelativeTime)
*
* returns a representation of the firing time that was last requested * for this Timer
*/

```

```
    public @StaticAnalyzable @CallerAllocatedResult @ScopedThis  
        HighResolutionTime getUserFireTime();  
}
```

---

## Unsigned.java

---

```
package javax.realtime.util.sc;  
  
/**  
 * This class has no analog in the RTSJ. It is provided as a notational convenience to encourage  
 * idiomatic usage with respect to the use and manipulation of unsigned quantities.  
 */  
public class Unsigned {  
  
    /**  
     * Compare arg1 with arg2. Return:  
     *  
     * ♦ -1 if first argument is smaller than second argument  
     * ♦ 0 if both arguments are equal  
     * ♦ 1 if first argument is larger than second argument  
     *  
     * All comparisons treat their arguments as if they are encoded according to unsigned integer  
     * conventions.  
     */  
    public static final @StaticAnalyzable int compare(byte arg1, byte arg2);  
  
    /**  
     * Compare arg1 with arg2. Return:  
     *  
     * ♦ -1 if first argument is smaller than second argument  
     * ♦ 0 if both arguments are equal  
     * ♦ 1 if first argument is larger than second argument  
     *  
     * All comparisons treat their arguments as if they are encoded according to unsigned integer  
     * conventions.  
     */  
    public static final @StaticAnalyzable int compare(short arg1, short arg2);  
  
    /**  
     * Compare arg1 with arg2. Return:  
     *  
     * ♦ -1 if first argument is smaller than second argument  
     * ♦ 0 if both arguments are equal  
     * ♦ 1 if first argument is larger than second argument  
     *  
     * All comparisons treat their arguments as if they are encoded according to unsigned integer  
     * conventions.  
     */  
    public static final @StaticAnalyzable int compare(int arg1, int arg2);  
  
    /**  
     * Compare arg1 with arg2. Return:
```

```
*
* ♦ -1 if first argument is smaller than second argument
* ♦ 0 if both arguments are equal
* ♦ 1 if first argument is larger than second argument
*
* All comparisons treat their arguments as if they are encoded according to unsigned integer
* conventions
*/
public static final @StaticAnalyzable int compare(long arg1, long arg2);

/**
 * Return true if first argument is greater than or equal to second argument, treating
 * arguments as if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean ge(byte arg1, byte arg2);

/**
 * Return true if first argument is greater than or equal to second argument, treating
 * arguments as if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean ge(short arg1, short arg2) {
    return false;
}

/**
 * Return true if first argument is greater than or equal to second argument, treating
 * arguments as if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean ge(int arg1, int arg2);

/**
 * Return true if first argument is greater than or equal to second argument, treating
 * arguments as if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean ge(long arg1, long arg2);

/**
 * Return true if first argument is greater than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean gt(byte arg1, byte arg2);

/**
 * Return true if first argument is greater than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean gt(short arg1, short arg2);

/**
 * Return true if first argument is greater than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean gt(int arg1, int arg2);
```

```
/**
 * Return true if first argument is greater than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean gt(long arg1, long arg2)

/**
 * Return true if first argument is less than or equal to second argument, treating arguments as
 * if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean le(byte arg1, byte arg2);

/**
 * Return true if first argument is less than or equal to second argument, treating arguments as
 * if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean le(short arg1, short arg2);

/**
 * Return true if first argument is less than or equal to second argument, treating arguments as
 * if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean le(int arg1, int arg2);

/**
 * Return true if first argument is less than or equal to second argument, treating arguments as
 * if encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean le(long arg1, long arg2);

/**
 * Return true if first argument is less than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean lt(byte arg1, byte arg2);

/**
 * Return true if first argument is less than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean lt(short arg1, short arg2);

/**
 * Return true if first argument is less than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean lt(int arg1, int arg2);

/**
 * Return true if first argument is less than the second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
```

```
public static final @StaticAnalyzable boolean lt(long arg1, long arg2);

/**
 * Return true if first argument is equal to second argument, treating arguments as if encoded
 * according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean eq(byte arg1, byte arg2);

/**
 * Return true if first argument is equal to second argument, treating arguments as if encoded
 * according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean eq(short arg1, short arg2);

/**
 * Return true if first argument is equal to second argument, treating arguments as if encoded
 * according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean eq(int arg1, int arg2);

/**
 * Return true if first argument is equal to second argument, treating arguments as if encoded
 * according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean eq(long arg1, long arg2);

/**
 * Return true if first argument is not equal to second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean neq(byte arg1, byte arg2);

/**
 * Return true if first argument is not equal to second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean neq(short arg1, short arg2);

/**
 * Return true if first argument is not equal to second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean neq(int arg1, int arg2);

/**
 * Return true if first argument is not equal to second argument, treating arguments as if
 * encoded according to unsigned integer conventions. Otherwise, return false.
 */
public static final @StaticAnalyzable boolean neq(long arg1, long arg2);

/**
 * Coerce argument to an unsigned 8-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 255.
```

```
*
* returns coerced result
*/
public static @StaticAnalyzable byte toByte(byte arg);

/**
* Coerce argument to an unsigned 8-bit quantity, throwing a previously allocated instance of
* UnsignedCoercionException if the number to be coerced is larger than 255.
*
* returns coerced result
*
* throws UnsignedCoercionException if the resulting type has too few bits to represent the
*         argument's unsigned value.
*/
public static @StaticAnalyzable byte toByte(short arg) throws UnsignedCoercionException;

/**
* Coerce argument to an unsigned 8-bit quantity, throwing a previously allocated instance of
* UnsignedCoercionException if the number to be coerced is larger than 255.
*
* returns coerced result
*
* throws UnsignedCoercionException if the resulting type has too few bits to represent the
*         argument's unsigned value.
*/
public static @StaticAnalyzable byte toByte(int arg) throws UnsignedCoercionException;

/**
* Coerce argument to an unsigned 8-bit quantity, throwing a previously allocated instance of
* UnsignedCoercionException if the number to be coerced is larger than 255.
*
* returns coerced result
*
* throws UnsignedCoercionException if the resulting type has too few bits to represent the
*         argument's unsigned value.
*/
public static @StaticAnalyzable byte toByte(long arg) throws UnsignedCoercionException;

/**
* Coerce argument to an unsigned 16-bit quantity, throwing a previously allocated instance of
* UnsignedCoercionException if the number to be coerced is larger than 65,535.
*
* returns coerced result
*/
public static @StaticAnalyzable short toShort(byte arg);

/**
* Coerce argument to an unsigned 16-bit quantity, throwing a previously allocated instance of
* UnsignedCoercionException if the number to be coerced is larger than 65,535.
*
* returns coerced result
*/
public static @StaticAnalyzable short toShort(short arg);
```

```
/**
 * Coerce argument to an unsigned 16-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 65,535.
 *
 * returns coerced result
 *
 * throws UnsignedCoercionException Thrown if the resulting type has too few bits to
 *         represent the argument's unsigned value.
 */
public static @StaticAnalyzable short toShort(int arg) throws UnsignedCoercionException;
```

```
/**
 * Coerce argument to an unsigned 16-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 65,535.
 *
 * returns coerced result
 *
 * throws UnsignedCoercionException Thrown if the resulting type has too few bits to
 *         represent the argument's unsigned value.
 */
public static @StaticAnalyzable short toShort(long arg) throws UnsignedCoercionException;
```

```
/**
 * Coerce argument to an unsigned 32-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 4,294,967,295.
 *
 * returns coerced result
 */
public static @StaticAnalyzable int toInt(byte arg);
```

```
/**
 * Coerce argument to an unsigned 32-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 4,294,967,295.
 *
 * returns coerced result
 */
public static @StaticAnalyzable int toInt(short arg);
```

```
/**
 * Coerce argument to an unsigned 32-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 4,294,967,295.
 *
 * returns coerced result
 */
public static @StaticAnalyzable int toInt(int arg);
```

```
/**
 * Coerce argument to an unsigned 32-bit quantity, throwing a previously allocated instance of
 * UnsignedCoercionException if the number to be coerced is larger than 4,294,967,295.
 *
 * returns coerced result
 */
```

```
* throws UnsignedCoercionException if the resulting type has too few bits to represent the
*     argument's unsigned value.
*/
public static @StaticAnalyzable int toInt(long arg) throws UnsignedCoercionException;

/**
 * Coerce argument to an unsigned 64-bit quantity
 *
 * returns coerced result
 */
public static @StaticAnalyzable long toLong(byte arg);

/**
 * Coerce argument to an unsigned 64-bit quantity
 *
 * returns coerced result
 */
public static @StaticAnalyzable long toLong(short arg);

/**
 * Coerce argument to an unsigned 64-bit quantity
 *
 * returns coerced result
 */
public static @StaticAnalyzable long toLong(int arg);

/**
 * Coerce argument to an unsigned 64-bit quantity
 *
 * returns coerced result
 */
public static @StaticAnalyzable long toLong(long arg);
}
```

---

---

## UnsignedCoercionException.java

---

```
package javax.realtime.util.sc;

/**
 * This class indicates an arithmetic overflow in coercing one unsigned value to another.
 */
public class UnsignedCoercionException extends java.lang.RuntimeException {

    /**
     * A constructor for UnsignedCoercionException.
     */
    public @StaticAnalyzable @ScopedThis UnsignedCoercionException();

    /**
     * A descriptive constructor for UnsignedCoercionException.
     */
}
```

```
*  
* Parameters:  
*  
* msg: Description of the error.  
*/  
public @StaticAnalyzable @ScopedPure UnsignedCoercionException(String msg);  
}
```

## 9. Standard Utility Libraries for Hard Real-Time Development

The `javax.realtime.util` package provides general purpose libraries similar to libraries available in `java.util`, with APIs that have been refactored to make them more appropriate for use in a scoped-memory, non-garbage collected environment. Design of hard real-time collections libraries are under way. As of the writing of the current draft of the document, our plan is to offer three different collections libraries. All three collection implementations are declared with the `@ReentrantScope` (or `@NestedReentrantScope`) annotation, so that internally allocated objects all reside at the same scope level.

1. Traditional `collection` uses checked assignments and will throw `IllegalAssignmentException` if anyone attempts to insert into the collection a value that resides in a more inner nested scope than the collection itself. We do not make copies of the newly inserted data.
2. `copyingcollection` requires that the types stored in the collection be `Reconstructable`. When a new value is inserted into the collection, it is reconstructed within the same reentrant scope that represents the collection.
3. `nestedcollection` is declared with the `@NestedReentrantScope` annotation. The implementation of these collections uses the byte-code verifier to enforce that all entries inserted into the collection reside within the same scope as the collection itself. Thus, newly installed entries do not need to be copied, and the types stored within the collection do not need to derive from `Reconstructable`.

The APIs will be provided in a future revision of this document, after the implementation has been completed.

---

---

### Delegate.java

---

```
package javax.realtime.util;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.RetentionPolicy;  
  
import java.lang.annotation.Retention;  
import java.lang.annotation.Target;  
import java.lang.annotation.Documented;  
  
@Documented @Target({ElementType.FIELD}) @Retention(RetentionPolicy.RUNTIME)  
public @interface Delegate  
{  
}
```

---

---

## DelegatedAnalyzable.java

---

```
package javax.realtime.util;

import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Documented;

import javax.realtime.util.sc.StaticLimit;

@Documented @Target({ElementType.METHOD}) @Retention(RetentionPolicy.RUNTIME)
public @interface DelegatedAnalyzable
{

    /**
     * The length of this array must be either 1 or must equal the number of
     * entries in the modes enumeration. Otherwise, the byte-code verifier will
     * reject this declaration. If the length of this array is 1, but the
     * enumeration has more than entry, the value of the single entry corresponds
     * to all analysis modes.
     *
     * If this attribute is true, the byte code verifier rejects as illegal any
     * method for which its analysis cannot derive upper bounds on execution time.
     */
    public boolean[] enforce_time_analysis() default {true};

    /**
     * The length of this array must be either 1 or must equal the number of
     * entries in the modes enumeration. Otherwise, the byte-code verifier will
     * reject this declaration. If the length of this array is 1, but the
     * enumeration has more than entry, the value of the single entry corresponds
     * to all analysis modes.
     *
     * If this code can be proven to be non-blocking, then it can be safely
     * invoked from within a context that holds an Atomic priority ceiling lock.
     * Otherwise, it should not be invoked from within such a context because
     * blocking would compromise the desired efficient PCP implementation.
     *
     * TRUE means we enforce non-blocking behavior. FALSE means we do not enforce
     * non-blocking behavior. FALSE does not necessarily mean that we prove
     * blocking behavior.
     */
    public boolean[] enforce_non_blocking() default {true};

    /**
     * The length of this array must be either 1 or must equal the number of
     * entries in the modes enumeration. Otherwise, the byte-code verifier will
     * reject this declaration. If the length of this array is 1, but the
     * enumeration has more than entry, the value of the single entry corresponds
```

```
* to all analysis modes.
*
* If this attribute is true, the byte code verifier rejects as illegal any
* method for which its analysis cannot derive upper bounds on stack growth or
* immortal memory consumption.
*
*/
public boolean[] enforce_memory_analysis() default {true};

/**
 * Developers may estimate CPU time requirements and provide this information
 * for use by other static analysis tools by initializing this attribute. This
 * practice is strongly discouraged because (1) developers may error in their
 * analysis and (2) the results of their analysis are very non-portable.
 *
 * @return
 */
long[] user_estimated_time() default {StaticLimit.UNANALYZED_TIME};

/**
 * Developers may estimate ImmortalMemory requirements and provide this information
 * for use by other static analysis tools by initializing this attribute. This
 * practice is strongly discouraged because (1) developers may error in their
 * analysis and (2) the results of their analysis are very non-portable.
 *
 * @return
 */
long[] user_estimated_heap() default {StaticLimit.UNANALYZED_SIZE_BYTES};

/**
 * Developers may estimate stack memory requirements and provide this information
 * for use by other static analysis tools by initializing this attribute. This
 * practice is strongly discouraged because (1) developers may error in their
 * analysis and (2) the results of their analysis are very non-portable.
 *
 * @return
 */
long[] user_estimated_stack() default {StaticLimit.UNANALYZED_SIZE_BYTES};

/**
 * When declaring a method or constructor to be analyzable, the developer may
 * identify multiple distinct characteristic behaviors. The results of
 * analysis are likely to depend on the current system state and/or parameter
 * values. The modes argument provides enumeration constants to represent each
 * of the distinct modes of analysis.
 *
 * The first element of the enumeration always represents the default mode,
 * which generally, but does not necessarily, represent the worst possible
 * resource requirements.
 */
java.lang.Class<? extends java.lang.Enum> modes()
    default StaticLimit.DefaultAnalysisMode.class;
```

```

/**
 * Represents the nanoseconds required to execute this method's code, in each
 * of the identified modes of operation.
 *
 * @see StaticLimit#UNANALYZED_TIME
 * @see StaticLimit#UNANALYZABLE_TIME
 */
long[] execution_time() default {};

/**
 * Represents the maximum number of bytes of stack memory required to execute
 * this method, including all methods that are invoked from this method.
 *
 * @see StaticLimit#UNANALYZED_SIZE_BYTES
 * @see StaticLimit#UNANALYZABLE_SIZE_BYTES
 */
long[] stack_bytes() default {};

/**
 * Represents the maximum number of bytes of "heap" memory required to execute
 * this method, including all methods that are invoked from this method. We
 * use the term "heap" loosely. This heap does not necessarily refer to Java's
 * garbage collected heap. Rather, it refers to the ImmortalMemory region.
 *
 * @see StaticLimit#UNANALYZED_SIZE_BYTES
 * @see StaticLimit#UNANALYZABLE_SIZE_BYTES
 */
long[] heap_bytes() default {};
}

```

---



---

## Delegator.java

---

```

package javax.realtime.util;

import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Documented;

@Documented @Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface Delegator
{
}

```

---



---

## Iterator.java

---

```

package javax.realtime.util;

```

```
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.Scoped;

public interface Iterator<E extends Reconstructable>
{
    @ScopedThis public boolean hasNext();
    @Scoped @ScopedThis public E next();
    @ScopedThis public void remove();
}

```

---

---

### Reconstructable.java

---

```
package javax.realtime.util;

/**
 * This interface serves a purpose similar to Cloneable, but it operates in a
 * mode more compatible with @Scoped memory assignment constraints. The
 * byte-code verifier assures that any class that implements the Reconstructable
 * interface has a public @StaticAnalyzable @ScopedPure constructor that
 * takes an argument of its own type.
 *
 * The idea is that the various collections classes require the object stores to
 * be implemented with Reconstructable objects. When a Reconstructable object is
 * inserted into a hash table, the data value is reconstructed within the hash
 * table's outer-nested scope in order to assure that the hash table data
 * structures can refer to it. Otherwise, it is likely that the object added
 * into the hash table lives in a more inner-nested scope than the hash table
 * itself, and this would result in an IllegalAssignmentException.
 */
public interface Reconstructable
{
}

```

## 10. Standard Hard Real-Time Extended API for Mission-Critical Java

Below, we describe each of the classes in the `javax.realtime.util.mc` (mission-critical) package. Since these classes are not described elsewhere, each class includes detailed commentary to describe the behavior of its supported methods.

---

---

### AllowCheckedScopedLinks.java

---

```
package javax.realtime.util.mc;

import java.lang.annotation.*;

/**
 * It is possible to build linked data structures out of stack-allocated objects. In the most

```

```

* general case, overwriting a reference field of one stack-allocated object to make it refer to
* another stack-allocated object requires a run-time check to assure that the referenced
* object is deeper on the stack (will survive at least as long as) than the referencing object.
*
* In the special case that the referencing object was just created within the most inner-nested
* activation frame, no run-time check is required. Thus, the @AllowCheckedScopedLinks
* annotation need not accompany constructors that initialize @Scoped reference instance
* variables. There is, however, an exception to this rule. If a constructor that initializes
* @Scoped reference instance variables is invoked to construct a caller-allocated return
* result object, this constructor must have the @AllowCheckedScopedLinks annotation.
* Otherwise, the code would be rejected by the safety-critical byte-code verifier. The reason
* to require checks in this case is that a "newly constructed" caller-allocated object is not
* necessarily the most shallow object on the run-time stack.
*
* In the more general case, the default configuration of safety-critical Java's byte-code
* verifier is to prohibit any assignments to stack-allocated objects that would possibly require
* a run-time check, and the possibility that a run-time exception would be thrown. To allow a
* particular method to build up a linked data structure out of stack-allocated objects using
* assignment operations that must be checked at run time, annotate the method with this
* annotation.
*
* In some projects, we expect project managers to prohibit all use of run-time checking
* associated with assignments to reference instance variables. We therefore strongly
* recommend a compiler and byte-code verification option that prohibits all use of the
* AllowCheckedScopedLinks annotation.
*/
@Documented @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME) public @interface AllowCheckedScopedLinks {
}

```

---



---

### CountingSemaphore.java

---

```

package javax.realtime.util.mc;

import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.StaticAnalyzable;

public class CountingSemaphore
{
    public @ScopedPure @StaticAnalyzable CountingSemaphore();
    public @ScopedPure @StaticAnalyzable void P();
    public @ScopedPure @StaticAnalyzable void V();
    public @ScopedPure @StaticAnalyzable int count();
}

```

---



---

### IllegalAssignmentException.java

---

```

package javax.realtime.util.mc;

import javax.realtime.util.sc.StaticAnalyzable;
import javax.realtime.util.sc.ScopedPure;

```

```
import javax.realtime.util.sc.ScopedThis;

/**
 * This is a checked exception to replace the RTSJ's unchecked IllegalAssignmentError. Any
 * method or constructor that carries the @AllowCheckedScopedLinks annotation is required to
 * declare as part of its signature that it throws this exception. This assures that a superclass
 * method not allowing checked scoped links is not overridden by a subclass method that does.
 */
public class IllegalAssignmentException extends java.lang.Exception {

    /**
     * Construct a new IllegalAssignmentException object.
     */
    public @StaticAnalyzable @ScopedThis IllegalAssignmentException();

    /**
     * Construct a new IllegalAssignmentException with the specified detail message.
     *
     * Parameters:
     *
     * msg: the detail message.
     */
    public @StaticAnalyzable @ScopedPure IllegalAssignmentException(String msg);
}
```

---

---

## Mutex.java

---

```
package javax.realtime.util.mc;

/**
 * The Mutex class implements priority-inheritance synchronization without
 * requiring LIFO access to locked resources.
 */
public class Mutex {
    public void enter();
    public void exit();
}
```

---

---

## OmitScopeChecking.java

---

```
package javax.realtime.util.mc;

/**
 * Method annotation to declare the programmer's desire to not enforce assignment
 * scope checking within the accompanying method's body.
 */
public @Documented @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
    @Retention(RetentionPolicy.RUNTIME) @interface OmitScopeChecking {
}
```

---

---

### OmitSubscriptChecking.java

---

```
package javax.realtime.util.mc;

import java.lang.annotation.*;

/**
 * Method annotation to declare the programmer's desire to not enforce array subscript
 * checking within that method's body.
 *
 * See Configuration.ELIDE_ARRAY_SUBSCRIPT_CHECKING
 */
@Documented @Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME) public @interface OmitSubscriptChecking {
}
```

---

---

### PreallocatedExceptions.java

---

```
package javax.realtime.util.mc;

public class PreallocatedExceptions {
    public static final IllegalArgumentException IllegalAssignmentException =
        new IllegalArgumentException();
}
```

---

---

### ReentrantScope.java

---

```
package javax.realtime.util.mc;

import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Documented;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

/**
 * Annotate a class with the @ReentrantScope attribute in order to force that each virtual
 * method's memory allocation take its memory from a common Scope that is allocated at the
 * time the class instance is constructed.
 */
@Documented @Target({ElementType.TYPE}) @Inherited
@Retention(RetentionPolicy.RUNTIME) public @interface ReentrantScope
{
}
```

---

---

## Registry.java

---

```
package javax.realtime.util.mc;

import javax.realtime.util.sc.Scoped;
import javax.realtime.util.sc.ScopedThis;
import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.mc.AllowCheckedScopedLinks;

final public class Registry {

    /**
     * Return a reference to the primordial instance of the registry.
     *
     * throws IllegalStateException if the primordial instance has not yet been instantiated.
     */
    public static @Scoped Registry instance() throws IllegalStateException;

    /**
     * returns a reference to the registry instance that corresponds to the interface with the
     * traditional Java virtual machine known by name.
     *
     * throws IllegalStateException if no Registry has been instantiated to represent an interface
     * to the named virtual
     * machine.
     */
    public static @Scoped Registry instance(@Scoped String name) throws IllegalStateException;

    /**
     * Initialize the primordial Registry with the specified number of proxy stacks, each
     * containing stack_size bytes of memory. If the primordial Registry object has already been
     * initialized, this throws IllegalStateException.
     *
     * throws IllegalStateException if the primordial Registry object has already been
     * instantiated.
     */
    public Registry(int num_proxy_stacks, int stack_size) throws IllegalStateException;

    /**
     * Instantiate a Registry to represent the interface to the virtual machine identified by name,
     * with the specified number of proxy stacks, each containing stack_size bytes of memory.
     *
     * throws IllegalStateException if the Registry object corresponding to the named virtual
     * machine has already been initialized.
     */
    public Registry(String name, int num_proxy_stacks, int stack_size)
        throws IllegalStateException;

    /**
     * Publish a hard real-time object so that it can be shared with the traditional Java
     * environment.
     *
     */
}
```

```
*
* Parameters:
*
* name: The name by which this object will be known in the traditional Java lookup service.
*
* obj: The object to be published for sharing with the traditional Java domain.
*
* throws IllegalStateException if an object with this same name has already been published
*
* throws IllegalArgumentException if the scope containing object was not annotated with a
*     @TraditionalJavaShared attribute that corresponds to the same traditional Java
*     virtual machine as this Registry object, or if the scope of name is not assignable to
*     obj. (We need to assure that the lifetime of name is no shorter than the lifetime of
*     obj.)
*/
public final @ScopedPure void publish(String name, Object obj)
    throws IllegalStateException, IllegalArgumentException;

/**
 * Remove a hard real-time object from the public registry so that the traditional Java
 * environment can no longer look it up. Note that the traditional Java environment may still
 * have a way to see this object, even if it's no longer in the public registry. For example, a
 * traditional Java thread may have looked up the object before it was unpublished and
 * retained a reference to the object even after the object was removed from the registry.
 *
 * Parameters:
 *
 * obj: The object to be removed from the registry
 *
 * throws IllegalStateException if this object does not currently reside within the registry
 */
public final @ScopedPure void unpublish(Object obj) throws IllegalStateException;

/**
 * Remove a hard real-time object from the public registry so that the traditional Java
 * environment can no longer look it up. Note that the traditional Java environment may still
 * have a way to see this object, even if it's no longer in the public registry. For example, a
 * traditional Java thread may have looked up the object before it was unpublished and
 * retained a reference to the object even after the object was removed from the registry.
 *
 * Parameters:
 *
 * name: The current registry name of the object to be removed from the registry
 *
 * throws IllegalStateException if this object does not currently reside within the registry
 */
public final @ScopedPure void unpublish(String name) throws IllegalStateException;

/**
 * This routine is normally called from a method that has declared itself to have the
 * @TraditionalJavaShared attribute. If caller does not have this attribute, the method
 * returns immediately. If the caller does have this attribute, the method returns only
 * after all of the proxy objects that have been shared from this method's allocation context
```

```

* are all removed from the traditional Java domain. By default, this requires that the
* traditional Java garbage collector has reclaimed all of the proxy objects and the
* finalizer code for those proxy objects has unregistered the objects from the registry.
*
* Note that removal of objects from a shared Java registry should be a very rare event.
* Thus, we are willing to invest some "time and energy" in making sure there is a clean
* separation of concerns before destroying the corresponding hard real-time allocation
* context.
*/
public final @ScopedThis void awaitClearRegistry();

/**
 * Add num_stacks ThreadStacks to the pool of proxy threads associated with this Registry
 * object. The size of each thread's ThreadStack is the same size as was specified in the
 * constructor for this object.
 *
 * returns true if the stacks were successfully added, false otherwise.
 */
public @ScopedThis boolean addProxyThreadStacks(int num_stacks);
}

```

---

### SignalingSemaphore.java

---

```

package javax.realtime.util.mc;

import javax.realtime.util.sc.ScopedPure;
import javax.realtime.util.sc.StaticAnalyzable;

public class SignalingSemaphore
{
    public @ScopedPure @StaticAnalyzable SignalingSemaphore();
    public @ScopedPure @StaticAnalyzable void P();
    public @ScopedPure @StaticAnalyzable void V();
    public @ScopedPure @StaticAnalyzable void Vall();
}

```

---

### ThreadStack.java

---

```

package javax.realtime.util.mc;

import javax.realtime.SchedulingParameters;
import javax.realtime.util.mc.AllowCheckedScopedLinks;
import javax.realtime.MemoryArea;
import javax.realtime.util.sc.Scoped;

import javax.realtime.util.sc.PreallocatedExceptions;

final public class ThreadStack extends javax.realtime.LTMemory {

    public @ScopedThis ThreadStack(long byte_size);
}

```

```
/**
 * This spawn method is used to start up a NoHeapRealtimeThread within the memory scope
 * represented by this ThreadStack object.
 *
 * It requires that its the_class argument extend from NoHeapRealtimeThread. It also
 * requires that the_class have a constructor that takes as arguments an arbitrary @Scoped
 * Object, a @Scoped SchedulingParameters, and a @Scoped MemoryArea. The second and
 * third of these parameters are presumed to be passed to the super-class constructor for
 * NoHeapRealtimeThread. If the_class does not extend from NoHeapRealtimeThread or
 * does not have a constructor with the desired signature, spawn() terminates by throwing an
 * IllegalArgumentException.
 *
 * The byte-code verifier assures that this spawn() invocation is matched by a join() in the
 * corresponding finally clause. Invocation of spawn() establishes the scope context at
 * which subsequent joinAndSpawn() invocations will be anchored.
 *
 * Parameters:
 *
 * the_class: The sub-class of NoHeapRealtimeThread that will be instantiated and started
 * within the memory scope represented by this ThreadStack object.
 *
 * argument: The argument that will be passed in the first position to the constructor for
 * the_class.
 *
 * parm: The SchedulingParameters argument that governs the scheduling of the newly
 * spawned thread.
 *
 * throws IllegalArgumentException if the_class does not extend NoHeapRealtimeThread or
 * does not have a constructor expecting a single Object argument.
 *
 * throws IllegalStateException if this ThreadStack already executed spawn() but not the
 * corresponding join().
 */
public @ScopedPure synchronized
    void spawn(Class the_class, Object argument, SchedulingParameters parm)
        throws IllegalArgumentException, IllegalStateException;
}
```

```
/**
 * This joinAndSpawn() method is used to first wait for a previously spawned thread to
 * terminate its execution, and then start up a new NoHeapRealtimeThread within the memory
 * scope represented by this ThreadStack object, overwriting the memory that had been
 * previously dedicated to execution of the thread that was previously running.
 *
 * It requires that its the_class argument extend from NoHeapRealtimeThread. It also
 * requires that the_class have a constructor that takes as arguments an arbitrary @Scoped
 * Object, a @Scoped SchedulingParameters, and a @Scoped MemoryArea. The second and
 * third of these parameters are presumed to be passed to the super-class constructor for
 * NoHeapRealtimeThread. If the_class does not extend from NoHeapRealtimeThread or
 * does not have a constructor with the desired signature, spawn() terminates by throwing an
 * IllegalArgumentException.
 *
 */
```

```

* Parameters:
*
* the_class: The sub-class of NoHeapRealtmeThread that will be instantiated and started
*   within the memory scope represented by this ThreadStack object.
*
* argument: The argument that will be passed in the first position to the constructor for
*   the_class.
*
* parm: The SchedulingParameters argument that governs the scheduling of the newly
*   spawned thread.
*
* throws IllegalArgumentException if the_class does not extend NoHeapRealtmeThread or
*   does not have a constructor expecting a single Object argument.
*
* throws IllegalStateException if this ThreadStack has never executed spawn() or if the
*   most recent spawn() invocation was already matched by a corresponding join().
*
* throws IllegalAssignmentException if any of the arguments are not assignment compatible
*   with the scope from which this ThreadStack was most recently spawned. In
*   particular, referenced @Scoped objects must reside within a context that is
*   either the same as or more outer-nested than the activation frame from which this
*   method was most recently spawned.
*/
public @AllowCheckedScopedLinks @ScopedPure synchronized
    void joinAndSpawn(Class the_class, Object argument, SchedulingParameters parm)
        throws IllegalArgumentException, IllegalStateException, IllegalAssignmentException;
}

/**
* This spawn method is used to start up a NoHeapRealtmeThread within the memory scope
* represented by this ThreadStack object.
*
* It requires that its the_class argument extend from NoHeapRealtmeThread. It also
* requires that the_class have a constructor that takes as arguments a @Scoped
* SchedulingParameters and a @Scoped MemoryArea. These two parameters are presumed
* to be passed to the super-class constructor for NoHeapRealtmeThread. If the_class does
* not extend from NoHeapRealtmeThread or does not have a constructor with the desired
* signature, spawn() terminates by throwing an IllegalArgumentException.
*
* The byte-code verifier assures that this spawn() invocation is matched by a join() in the
* corresponding finally clause. Invocation of spawn() establishes the scope context at
* which subsequent joinAndSpawn() invocations will be anchored.
*
* Parameters:
*
* the_class: The sub-class of NoHeapRealtmeThread that will be instantiated and started
*   within the memory scope represented by this ThreadStack object.
*
* parm: The SchedulingParameters argument that governs the scheduling of the newly
*   spawned thread.
*
* throws IllegalArgumentException if the_class does not extend NoHeapRealtmeThread or

```

```

*      does not have a constructor expecting a single Object argument.
*
* throws IllegalStateException if this ThreadStack already executed spawn() but not the
*      corresponding join().
*/
public @AllowCheckedScopedLinks @ScopedPure synchronized
    void spawn(Class the_class, SchedulingParameters parm)
        throws IllegalArgumentException, IllegalStateException {
}

/**
* This joinAndSpawn() method is used to first wait for a previously spawned thread to
* terminate its execution, and then start up a new NoHeapRealtimeThread within the memory
* scope represented by this ThreadStack object, overwriting the memory that had been
* previously dedicated to execution of the thread that was previously running.
*
* It requires that its the_class argument extend from NoHeapRealtimeThread. It also
* requires that the_class have a constructor that takes as arguments a @Scoped
* SchedulingParameters and a @Scoped MemoryArea. These two parameters are presumed
* to be passed to the super-class constructor for NoHeapRealtimeThread. If the_class does
* not extend from NoHeapRealtimeThread or does not have a constructor with the desired
* signature, spawn() terminates by throwing an IllegalArgumentException.
*
* Parameters:
*
* the_class: The sub-class of NoHeapRealtimeThread that will be instantiated and started
*      within the memory scope represented by this ThreadStack object.
*
* parm: The SchedulingParameters argument that governs the scheduling of the newly
*      spawned thread.
*
* parm: The SchedulingParameters argument that governs the scheduling of the newly
*      spawned thread.
*
* throws IllegalArgumentException if the_class does not extend NoHeapRealtimeThread or
*      does not have a constructor expecting no arguments.
*
* throws IllegalStateException if this ThreadStack has never executed spawn() or if the
*      most recent spawn() invocation was already matched by a corresponding join().
*
* throws IllegalAssignmentException if any of the arguments are not assignment compatible
*      with the scope from which this ThreadStack was most recently spawned. In
*      particular, referenced @Scoped objects must reside within a context that is
*      either the same as or more outer-nested than the activation frame from which this
*      method was most recently spawned.
*/
public @AllowCheckedScopedLinks @ScopedPure synchronized
    void joinAndSpawn(Class the_class, SchedulingParameters parm)
        throws IllegalArgumentException, IllegalStateException, IllegalAssignmentException;
}

/**
* This join() method is used to wait for a previously spawned thread to terminate its

```

```
* terminate its execution.
*
* throws IllegalStateException if this ThreadStack has never executed spawn() or if the
*     most recent spawn() invocation was already matched by a corresponding join().
*/
public @ScopedThis synchronized void join() throws IllegalStateException;
}
```

---

---

### TraditionalJavaMethod.java

---

```
package javax.realtime.util.mc;

import java.lang.annotation.*;

@Documented @Target({ElementType.METHOD}) @Retention(RetentionPolicy.CLASS)
    @Inherited public @interface TraditionalJavaMethod {
}
```

---

---

### TraditionalJavaShared.java

---

```
package javax.realtime.util.mc;

import java.lang.annotation.*;

@Documented @Target({ElementType.METHOD}) @Retention(RetentionPolicy.RUNTIME)
    @Inherited public @interface TraditionalJavaShared {

    /**
     * Name the virtual machine with which objects allocated in this allocation context can be
     * shared.
     */
    public String jvm_id() default "";

}
```

**Appendix D: Index of Program Library Components**

J	
java.io.Serializable	65
java.lang.annotation.Annotation	61
java.lang.annotation.AnnotationInfo	61
java.lang.annotation.Documented	61
java.lang.annotation.ElementType	61
java.lang.annotation.Inherited	62
java.lang.annotation.Overrides	62
java.lang.annotation.Retention	62
java.lang.annotation.RetentionPolicy	62
java.lang.annotation.Target	62
java.lang.ArithmeticException	38
java.lang.ArrayIndexOutOfBoundsException	38
java.lang.ArrayStoreException	38
java.lang.AssertionError	39
java.lang.Boolean	39
java.lang.Byte	39
java.lang.Character	40
java.lang.Class	42
java.lang.ClassCastException	42
java.lang.ClassFormatError	43
java.lang.Cloneable	44
java.lang.CloneNotSupportedException	44
java.lang.Comparable	45
java.lang.Double	45
java.lang.Enum	46
java.lang.Error	46
java.lang.Exception	47
java.lang.Float	47
java.lang.IllegalArgumentException	48
java.lang.IllegalMonitorStateException	48
java.lang.IllegalStateException	48
java.lang.IndexOutOfBoundsException	48
java.lang.Integer	49
java.lang.InterruptedIOException	50
java.lang.LinkageError	50
java.lang.Long	51
java.lang.Math	51
java.lang.NegativeArraySizeException	52
java.lang.NoSuchMethodException	53
java.lang.NullPointerException	53
java.lang.Number	53
java.lang.NumberFormatException	53
java.lang.Object	54

java.lang.OutOfMemoryError	55
java.lang.reflect.AccessibleObject	63
java.lang.reflect.AnnotatedElement	63
java.lang.reflect.Constructor	64
java.lang.reflect.Field	64
java.lang.reflect.Member	64
java.lang.reflect.Method	65
java.lang.Runnable	55
java.lang.RuntimeException	55
java.lang.SecurityException	55
java.lang.Short	56
java.lang.StackOverflowError	56
java.lang.StackTraceElement	57
java.lang.StringBuilder	58
java.lang.StringIndexOutOfBoundsException	59
java.lang.Thread	59
java.lang.Throwable	59
java.lang.UndeclaredThrowableException	60
java.lang.UnsupportedOperationException	60
java.lang.VirtualMachineError	60
java.util.Random	65
javax.realtime.AbsoluteTime	66
javax.realtime.AperiodicParameters	67
javax.realtime.ArrivalTimeQueueOverflowException	67
javax.realtime.AsyncEvent	68
javax.realtime.AsyncEventHandler	68
javax.realtime.BoundAsyncEventHandler	68
javax.realtime.Clock	69
javax.realtime.HighResolutionTime	69
javax.realtime.ImmortalMemory	70
javax.realtime.LTMemory	70
javax.realtime.MemoryArea	70
javax.realtime.MemoryParameters	71
javax.realtime.MITViolationException	70
javax.realtime.MonitorControl	71
javax.realtime.NoHeapRealtimeThread	71
javax.realtime.OneShotTimer	72
javax.realtime.PeriodicParameters	72
javax.realtime.PeriodicTimer	72
javax.realtime.PriorityCeilingEmulation	73
javax.realtime.PriorityInheritance	73
javax.realtime.PriorityParameters	73
javax.realtime.PriorityScheduler	74
javax.realtime.RealtimeThread	74
javax.realtime.RelativeTime	74

javax.realtime.ReleaseParameters	75
javax.realtime.Schedulable	75
javax.realtime.Scheduler	76
javax.realtime.SchedulingParameters	76
javax.realtime.ScopedMemory	76
javax.realtime.SizeEstimator	76
javax.realtime.SporadicParameters	77
javax.realtime.Timer	77
javax.realtime.UnknownHappeningException	78
javax.realtime.util.Delegate	208
javax.realtime.util.DelegatedAnalyzable	209
javax.realtime.util.Delegator	211
javax.realtime.util.Iterator	212
javax.realtime.util.mc.AllowCheckedScopedLinks	213
javax.realtime.util.mc.CountingSemaphore	213
javax.realtime.util.mc.IllegalAssignmentException	214
javax.realtime.util.mc.Mutex	214
javax.realtime.util.mc.OmitScopeChecking	214
javax.realtime.util.mc.OmitSubscriptChecking	215
javax.realtime.util.mc.PreallocatedExceptions	215
javax.realtime.util.mc.Registry	216
javax.realtime.util.mc.ThreadStack	218
javax.realtime.util.mc.TraditionalJavaMethod	222
javax.realtime.util.mc.TraditionalJavaShared	222
javax.realtime.util.Reconstructable	212
javax.realtime.util.sc.AbsoluteTime	121
javax.realtime.util.sc.AperiodicParameters	124
javax.realtime.util.sc.AsyncEvent	126
javax.realtime.util.sc.Atomic	127
javax.realtime.util.sc.BindAsyncEventHandler	128
javax.realtime.util.sc.CallerAllocatedArrayResult	129
javax.realtime.util.sc.CallerAllocatedResult	130
javax.realtime.util.sc.Ceiling	130
javax.realtime.util.sc.Clock	131
javax.realtime.util.sc.Configuration	132
javax.realtime.util.sc.EmbeddedConflictException	138
javax.realtime.util.sc.FinalSizeEstimator	139
javax.realtime.util.sc.HighResolutionTime	142
javax.realtime.util.sc.ImmortalAllocation	155
javax.realtime.util.sc.InitializeAtStartup	156
javax.realtime.util.sc.InterruptEvent	156
javax.realtime.util.sc.InterruptHandler	158
javax.realtime.util.sc.IOPort	143
javax.realtime.util.sc.IOPort16I	152
javax.realtime.util.sc.IOPort16IO	153

javax.realtime.util.sc.IOPort16O	153
javax.realtime.util.sc.IOPort32I	153
javax.realtime.util.sc.IOPort32IO	153
javax.realtime.util.sc.IOPort32O	154
javax.realtime.util.sc.IOPort64I	154
javax.realtime.util.sc.IOPort64IO	154
javax.realtime.util.sc.IOPort64O	154
javax.realtime.util.sc.IOPort8I	155
javax.realtime.util.sc.IOPort8IO	155
javax.realtime.util.sc.IOPort8O	155
javax.realtime.util.sc.OneShotTimer	163
javax.realtime.util.sc.PCP	164
javax.realtime.util.sc.PeriodicParameters	165
javax.realtime.util.sc.PeriodicTimer	167
javax.realtime.util.sc.PreallocatedExceptions	168
javax.realtime.util.sc.RelativeTime	170
javax.realtime.util.sc.ReleaseParameters	175
javax.realtime.util.sc.Scoped	177
javax.realtime.util.sc.ScopedArray	177
javax.realtime.util.sc.ScopedArrayLocal	177
javax.realtime.util.sc.ScopedLocal	178
javax.realtime.util.sc.ScopedMemorySize	178
javax.realtime.util.sc.ScopedPure	179
javax.realtime.util.sc.ScopedThis	180
javax.realtime.util.sc.ScopedThisLocal	180
javax.realtime.util.sc.SignalingSemaphore	218
javax.realtime.util.sc.SizeEstimator	181
javax.realtime.util.sc.SporadicParameters	183
javax.realtime.util.sc.StaticDependency	187
javax.realtime.util.sc.StaticLimit	188
javax.realtime.util.sc.TimeBuffer	196
javax.realtime.util.sc.Timer	199
javax.realtime.util.sc.Unsigned	201
javax.realtime.util.sc.UnsignedCoercionException	207