

The Real-Time Specification for Java™

Version 1.0.2

Editor: Peter Dibble

Authors

Rudy Belliardi Ben Brosgol
Peter Dibble David Holmes Andy Wellings

Other Contributors

Maintenance Lead: Peter Dibble
Former maintenance leads: Doug Locke, Peter Haggar
Initial Spec Lead: Greg Bollella

Authors of the First Edition

The Real-Time For Java Expert Group

Greg Bollella	Ben Brosgol
Peter Dibble	Steve Furr
James Gosling	David Hardin
Mark Turnbull	Rudy Belliardi

The Reference Implementation Team

Doug Locke	Scott Robbins
Pratik Solanki	Dionisio de Niz



Copyright © 2000, 2003, 2004, 2005, 2006 TimeSys Corp.

Duke logo TM designed by Joe Palrang.

Sun, Sun Microsystems, the Sun logo, the Duke logo, and all Sun, Java, Jini, and Solaris based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All other product names mentioned herein are the trademarks of their respective owners.

U.S. GOVERNMENT USE: This specification relates to commercial items, processes or software. Accordingly, use by the United States Government is subject to these terms and conditions, consistent with FAR12.211 and 12.212.

TIME SYS CORPORATION HEREBY GRANTS TO YOU A FULLY PAID, NONEXCLUSIVE, NONTRANSFERABLE WORLDWIDE, LIMITED LICENSE (WITHOUT THE RIGHT TO SUBLICENSE), UNDER TIME SYS' APPLICABLE INTELLECTUAL PROPERTY RIGHTS TO VIEW, DOWNLOAD, USE AND REPRODUCE THE SPECIFICATION ONLY FOR THE PURPOSE OF INTERNAL EVALUATION. THIS INCLUDES (I) DEVELOPING APPLICATIONS INTENDED TO RUN ON AN IMPLEMENTATION OF THE SPECIFICATION, PROVIDED THAT SUCH APPLICATIONS DO NOT THEMSELVES IMPLEMENT ANY PORTIONS(S) OF THE SPECIFICATION, AND (II) DISCUSSING THE SPECIFICATION WITH ANY THIRD PARTY; AND (III) EXCERPTING BRIEF PORTIONS OF THE SPECIFICATION IN ORAL OR WRITTEN COMMUNICATIONS WHICH DISCUSS THE SPECIFICATION PROVIDED THAT SUCH EXCERPTS DO NOT IN THE AGGREGATE CONSTITUTE AN SIGNIFICANT PORTION OF THE SPECIFICATION.

LICENSE TERMS APPROPRIATE FOR IMPLEMENTATION OF THE SPECIFICATION MAY BE FOUND ON-LINE AT — www.rtsj.org/specjavadoc/book_index.html, AND ARE AVAILABLE ON REQUEST FROM TIME SYS CORPORATION OR THE JAVA COMMUNITY PROCESS (JCP.ORG).

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION.

Contents

Authors	7
Foreword to the Second Edition	11
Foreword to the First Edition	13
Preface to the Second Edition	17
Preface to the First Edition	35
1 Introduction	1
2 Design	5
3 Requirements and Conventions	17
4 Standard Java Classes	23
5 Real-Time Threads	27
RealtimeThread	29
NoHeapRealtimeThread	55
6 Scheduling	59
Schedulable	81
Scheduler	97
PriorityScheduler	103
SchedulingParameters	112
PriorityParameters	113
ImportanceParameters	115
ReleaseParameters	116
PeriodicParameters	123
AperiodicParameters	130
SporadicParameters	137
ProcessingGroupParameters	144
7 Memory Management	153
MemoryArea	163
HeapMemory	170
ImmortalMemory	170
SizeEstimator	171
ScopedMemory	174
LTMemory	189
VTMemory	195
PhysicalMemoryManager	199
PhysicalMemoryTypeFilter	207

ImmortalPhysicalMemory	214
LTPhysicalMemory	223
VTPhysicalMemory	232
RawMemoryAccess	241
RawMemoryFloatAccess	264
MemoryParameters	275
GarbageCollector	280
8 Synchronization	283
MonitorControl	288
PriorityCeilingEmulation	290
PriorityInheritance	293
WaitFreeWriteQueue	294
WaitFreeReadQueue	299
WaitFreeDequeue	305
9 Time	311
HighResolutionTime	316
AbsoluteTime	322
RelativeTime	334
RationalTime	344
10 Clocks and Timers	349
Clock	354
Timer	356
OneShotTimer	371
PeriodicTimer	373
11 Asynchrony	381
AsyncEvent	390
AsyncEventHandler	395
BoundAsyncEventHandler	421
Interruptible	423
AsynchronouslyInterruptedException	424
Timed	430
12 System and Options	433
POSIXSignalHandler	434
RealtimeSecurity	441
RealtimeSystem	443
13 Exceptions	447
ArrivalTimeQueueOverflowException	448
CeilingViolationException	449
DuplicateFilterException	449
IllegalAssignmentError	450
InaccessibleAreaException	451

MemoryAccessError	451
MemoryInUseException	452
MemoryScopeException	453
MemoryTypeConflictException	453
MITViolationException	454
OffsetOutOfBoundsException	455
ScopedCycleException	456
SizeOutOfBoundsException	456
ThrowBoundaryError	457
UnsupportedPhysicalMemoryException	458
UnknownHappeningException	458
ResourceLimitError	459
14 Almanac	463
Index	495

Unofficial

Unofficial

Authors

Authors of the first edition

Greg Bollella, a Distinguished Engineer and Principle Investigator for Real-Time Java at Sun Microsystems Laboratories, was the founding Spec Lead for the RTSJ while a Senior Architect at IBM. He holds a Ph.D. in computer science from the University of North Carolina at Chapel Hill. His dissertation research is in real-time scheduling theory and real-time systems implementation.

Ben Brosgol is a senior technical staff member of Ada Core Technologies, Inc. He has had a long involvement with programming language design and implementation, focusing on Ada and real-time support, and has been providing Java-related services since 1997. Ben holds a Ph.D. in applied mathematics from Harvard University and a B.A. from Amherst College.

Peter Dibble, was the Senior Scientist at Microware Systems Corporation throughout the initial design of the RTSJ and nearly to the final publication of the 1.0 version of the RTSJ. At Microware he designed, coded, and analyzed system software for real-time systems for more than twelve years and as part of Microware's Java team, Peter was involved with the Java Virtual Machine since early 1997. He has a Ph.D. and M.S. in Computer Science from University of Rochester.

Steve Furr currently works for QNX Software Systems, where he was responsible for Java technologies for the QNX Neutrino Operating System. He graduated from Simon Fraser University with a B.Sc. in computer science.

James Gosling, a Fellow at Sun Microsystems, is the originator of the Java programming language. His career in programming started by developing real-time software for scientific instrumentation. He has a Ph.D. and M.Sc. in computer science from Carnegie-Mellon University and a B.Sc. in computer science from the University of Calgary.

David Hardin, Chief Technical Officer and co-founder of aJile Systems, has worked in safety-critical computer systems architecture, formal methods, and custom microprocessor design at Rockwell Collins, and was named a Rockwell Engineer of the Year for 1997. He holds a Ph.D. in electrical and computer engineering from Kansas State University.

Mark Turnbull has been an employee of Nortel Networks since 1983. Most of his experience has been in the area of proprietary language design, compiler design, and real-time systems.

Rudy Belliardi is a Consulting Engineer, Advanced Technology at Schneider Automation. He has designed and implemented systems, languages and protocols for factory automation and for the medical field. He holds a Doctor in electronic engineering from the University of Genova and an Applied Scientist Professional degree in computer science from the George Washington University.

Authors of the second edition

The bulk of the work on this version of the specification divided among the active members of the RTSJ Technical Interpretation Committee. The modifications were negotiated with Peter Dibble, then checked and approved by the entire group. Although each author has particular responsibility for certain chapters, the entire specification is a joint work. The major contributors are listed here.

Rudy Belliardi is a Consulting Engineer, Advanced Technology at Schneider Automation. He has designed and implemented systems, languages and protocols for factory automation and for the medical field. He is actively involved in industrial real-time communications efforts. He holds a Doctor in electronic engineering from the University of Genova and an Applied Scientist Professional degree in computer science from the George Washington University. He was the primary contributor for the Time and Clocks and Timers chapters.

Ben Brosgol is a senior technical staff member of Ada Core Technologies, Inc. He has had a long involvement with programming language design and implementation, focusing on Ada and real-time support, and has been providing Java-related services since 1997. Ben holds a Ph.D. in applied mathematics from Harvard University and a B.A. from Amherst College. He was the primary contributor for the Synchronization chapter.

Peter Dibble, Distinguished Engineer at TimeSys, is on the team that implemented and now supports the RTSJ reference implementation, the RTSJ technology conformance kit, and the first commercial implementation of the RTSJ. He acted as the technical lead for RTSJ spec interpretation and maintenance since TimeSys assumed the RTSJ maintenance lead role, and moved to Maintenance Lead late in the process. He served as editor for this edition, was the primary contributor to the Memory chapter, and was part of the team for the scheduling and threads chapters.

David Holmes is Director and Chief Scientist of DLTeCH Pty Ltd, located in Brisbane, Australia. His work with Java technology has focused on concurrency and synchronization support in the language and virtual machine and he is currently working on a real-time Java virtual machine. David is a member of the expert group for JSR-166 “Concurrency Utilities” being developed under the Java Community Process, and co-author of “The Java Programming Language” - third and fourth editions. David completed his Ph.D. at Macquarie University, Sydney, in 1999, in the area of synchronization within object-oriented systems. He was such an intense and

useful critic of the 1.0 RTSJ Spec and throughout this revision that the other authors asked him to help with this revision. He helped improve the specification everywhere, but contributed particularly heavily to the scheduling chapter.

Andy Wellings is Professor of Real-Time Systems in the Department of Computer Science, University of York, U.K. His research interests are focused on two related areas of computing: the design, use and implementation of real-time programming languages and operating systems; and the design and use of general purpose distributed operating systems. Professor Wellings has published over 100 technical papers and reports, including five textbooks. He teaches courses in Operating Systems, Real-Time Systems and Networks and Distributed Systems. He was a primary contributor to the Asynchrony chapter, and part of the team for the scheduling and threads chapters.

Other Contributors

Countless people have contributed to the RTSJ, but several are have committed so many hours to the effort that they stand out even in this hard-working group:

Peter Haggar, Senior Software Engineer at IBM, and a distinguished member of the Java community, assumed the role of RTSJ Spec Lead at IBM when Greg Bollella moved to Sun Microsystems. He saw the spec through public review and approval by the JCP Executive Committee, then handed the Maintenance Lead role to Doug Locke at TimeSys. He has a B.S in Computer Science from Clarkson University.

C. Douglass Locke, was Vice President of Technology at TimeSys, and Maintenance Lead for the RTSJ during most of the work on this specification. He managed the reference implementation and technology conformance kit projects at TimeSys for most of their progress. He has a Ph.D. from Carnegie-Mellon University and has worked in embedded real-time since computers ran from punch boards. Doug is currently an independent consultant.

The team that implemented the reference implementation contributed greatly to the specification. First, they separated the expert group's dreams from implementable reality. Second, by supplying an implementation that we could use, they uncovered serious problems with usability. Third, they became experts in the details of the specification and helped thrash out the final modifications to scoped memory and asynchronous transfer of control.

Three RI engineers stand out for their contributions: Dionisio deNizVillasenor Ph.D. student in Computer Science at Carnegie-Mellon University, Scott Robbins BS in Computer Science from Carnegie-Mellon University, and Pratik Solanki M.Sc. in Computer Science from Carnegie-Mellon University.

Alden Dima and the National Institute of Standards and Technology has supported the RTSJ mailing lists for years. This is nearly invisible support work, but it is important and we appreciate it.

Unofficial

Foreword to the Second Edition

In his Foreword to the first edition of the *Real-Time Specification for Java*, written before the RTSJ had been completed, Doug Jensen stated that he expected the RTSJ “to become the first real-time programming language to be both commercially and technologically successful.” Using the major advantage now accorded by hindsight, it seems clear that his optimism was not misplaced, even though his prediction cannot yet be completely verified.

Indeed, since its official release in 2001, the RTSJ and the community it has spawned has steadily and continuously matured to the point that there are now several viable implementations, many research papers, and a number of major real-time or embedded projects using the RTSJ. Thus, it now requires considerably less prescience to predict that the RTSJ will become the first real-time programming language to be both commercially and technologically successful.

It is interesting to consider that, as with many prior technology advances, the domain of applicability of the RTSJ has proven to be greater than its designers originally intended. Clearly, the RTSJ was targeted to real-time systems that can be described as systems whose correctness requires reasoning about their timing properties in addition to their functional properties, such as flight control or industrial sensor control systems. It has, moreover, now become clear that the RTSJ provides capabilities that greatly enhance Java’s use in embedded systems, regardless of whether they require real-time performance. Embedded systems can be described as systems whose owners and users are not concerned, and may even be unaware, that computers are present. Examples cover an extremely wide spectrum from cell phones to air traffic control systems. In general, real-time systems

Thus, it has now become clear that the RTSJ’s name does not adequately express its scope of applicability. The RTSJ is certainly targeted toward real-time systems by providing critical real-time capabilities such as explicit support for scheduling, bounded priority inversion, periodic `RealtimeThreads`, and even `NoHeapRealtimeThreads` for hard-real-time support. However, designers of embedded systems with no requirements, or only minimal requirements for real-time performance find the RTSJ useful because of capabilities such as `AsyncEvents` and `AsyncEventHandlers`, `Clocks`, `Timers`, and `POSIXSignalHandlers`. In fact, as implementers have increased their use of RTSJ-compliant platforms, they have discovered that handling POSIX/UNIX/Linux signals in RTSJ Java is far easier and far less error-prone than it has been in C or C++.

Change is a nearly universal property of successful standards. The only static standards are those that are not used. Thus, it is hardly surprising that the RTSJ has been continuously clarified, extended, and corrected since its initial release. Its maintainers, now under the direction of Peter Dibble from TimeSys, have carefully sought to ensure that existing applications and implementations will be unaffected or minimally affected by these updates, but such updates are a characteristic of all successful standards.

The RTSJ community is concurrently in the process of considering additional standards built on the RTSJ framework. The Open Group, in concert with several vendors and developers, is considering a Safety-Critical subset of Java based on the RTSJ. Several groups of developers are considering the creation of profiles of the RTSJ targeting specific application domains such as military command and control or consumer electronics.

As the RTSJ continues to mature, its community can be expected to further expand. At this point, it appears likely that the RTSJ will not only succeed, but it can be expected to make a significant difference in how real-time and embedded applications are designed, resulting in major improvements in responsiveness, maintainability, portability, and controlled development cost.

Indeed, it exactly these characteristics that have always been, and continue to be, the fundamental goals for all of the RTSJ's architects and contributors!

Doug Locke
Mount Lebanon, PA

Foreword to the First Edition

I expect *The Real-Time Specification for Java* to become the first real-time programming language to be both commercially and technologically successful.

Other programming languages have been intended for use in the real-time computing domain. However, none has been commercially successful in the sense of being significantly adopted in that domain. Many were academic research projects. Most did not focus on the core real-time issues of managing computing resources in order to satisfy application timeliness requirements. Instead, they typically emphasized the orthogonal (albeit important) topic of concurrency and other topics important to the whole field of embedded computing systems (of which real-time computing systems are a subset).

Ada 95, including its Real-Time Systems Annex D, has probably been the most successful real-time language, in terms of both adoption and real-time technology. One reason is that Ada is unusually effective (among real-time languages and also operating systems) across the real-time computing system spectrum, from programming-in-the-small in traditional device-level control subsystems, to programming-in-the-large in enterprise command and control systems. Despite that achievement, a variety of nontechnical factors crippled Ada's commercial success.

When James Gosling introduced the Java programming language in 1995, it appeared irrelevant to the real-time computing field, based on most of its initial purposes and its design. Indeed, some of its fundamental principles were antithetical to those of real-time computing. To facilitate its major goal of operating system and hardware independence, the language was deliberately given a weak vocabulary in areas such as thread behavior, synchronization, interrupts, memory management, and input/output. However, these are among the critical areas needing explicit management (by the language or the operating system) for meeting application timeliness requirements.

Nevertheless, the Java platform's promise of "Write Once, Run Anywhere," together with the Java language's appeal as a programming language *per se*, offer far greater cost-savings potential in the real-time (and more broadly, the embedded) domain than in the desktop and server domains. Desktops are dominated by the "Wintel" duopoly; servers have only a few processor types and operating systems. Real-time computing systems have tens of different processor types and many tens of different operating system products (not counting the custom-made ones that currently constitute about half of the installations). The POSIX standard hasn't provided the intended real-time application portability because it permits widely varying subsets to be implemented. The Java platform is already almost ubiquitous.

The real-time Java platform's necessarily qualified promise of "Write Once Carefully, Run Anywhere Conditionally" is nevertheless the best prospective opportunity for application re-usability.

The overall challenge was to reconcile the intrinsically divergent natures of the Java language and most of real-time computing. Compatibility of the Real-Time Specification for Java and the Java Language Specification had to be maintained, while making the former cost-effective for real-time computing systems.

Most people involved in, and even aware of, the real-time Java effort, including the authors of this book and me, were initially very skeptical about the feasibility of adequately meeting this challenge.

The real-time Java community took two important and unusual initial steps before forming the Real-Time for Java Expert Group under Sun's Java Community Process.

The first step was to convene many representatives of the real-time community a number of times (under the auspices of the National Institute for Standards and Technology), to achieve and document consensus on the requirements for the Real-Time Specification for Java. Not surprisingly, when this consensus emerged, it included mandatory requirements for building the kind of smaller scale, static, real-time subsystems familiar to current practitioners using C and C++.

More surprisingly, the consensus also included mandatory and optional requirements for accommodating advanced dynamic and real-time resource management technologies, such as asynchronous transfer of control and timeliness-based scheduling policies, and for building larger scale real-time systems. The primary impetus for these dynamic and programming-in-the-large, real-time requirements came from the communities already using the Java language, or using the Ada language, or building defense (primarily command and control) systems.

The second, concomitant, step was to establish an agreed-upon lexicon of real-time computing concepts and terms to enable this dialog about, and consensus on, the requirements for the Real-Time Specification for Java. As unlikely as it may seem to those outside of the real-time community, real-time computing concepts and terms are normally not used in a well-defined way (except by most real-time researchers).

The next step toward the realization of the Java language's potential for the present and the future of real-time computing is defining and writing the Real-Time Specification for Java, the first version of which is in this book. Understanding this specification will also improve the readers' understanding of both the Java language and real-time computing systems as well.

Greg Bollella was an ideal leader for this specification team. He recruited a well balanced group of real-time and Java language experts. His background in both practical and theoretical real-time computing prepared him for gently but resolutely guiding the team's rich and intense discussions into a coherent specification.

Of course, more work remains, including documenting use cases and examples; performing implementations and evaluations; gaining experience from deployed products; and iterations on *The Real-Time Specification for Java*. The Distributed Real-Time Specification for Java also lies ahead.

The real-time Java platform is prepared not just to provide cost-reduced functional parity with current mainstream real-time computing practice and products, but also to play a leadership role as real-time computing practice moves forward in the Internet age.

E. Douglas Jensen
Sherborn, MA

Unofficial

Unofficial

Preface to the Second Edition

The Evolution of the Specification

The RTSJ was first published in mid-2000, as RTSJ version 0.9. Version 0.9 was the result of the Expert Group's work between early 1999 and shortly before Java One in 2000. This document was not intended as a usable specification, but rather as a "draft for discussion."

Vigorous work on the specification continued after the publication of version 0.9, especially creation of the reference implementation and first use of the reference implementation. That work resulted in the first official version of RTSJ, which emerged from the Java Community Process (JCP) in January of 2002.

Work on the RTSJ hardly slowed when the specification was formally released. Several members of the Expert Group joined some interested members of the RTSJ community as the Technical Interpretation Committee (TIC) and continued to improve the specification.

The RTSJ TIC	
Ben Brosgol	Ada Core
David Holmes	DLTeCH Pty
Rudy Belliardi	Schneider Automation
Doug Locke	TimeSys
Peter Dibble	TimeSys
Andy Wellings	University of York

The main product of the TIC's work was RTSJ version 1.0.1, a much more complete version of the specification. The amount of additional detail is indicated by the increase in the size of the printed document from around 250 pages to around 500 pages between RTSJ version 1.0 and 1.0.1. The revised specification emerged from the JCP maintenance review process in March 2005, and the TIC continued work.

This document is RTSJ version 1.0.2. It continues the interpretation work of 1.0.1 with many of the changes between 1.0.1. and 1.0.2 motivated by requests for clarification by RTSJ implementors and users.

The next stage of the RTSJ's evolution is a new JSR. The Expert Group for RTSJ version 1.1 has already been formed, and the proposal for RTSJ version 1.1 can be found on the Java Community Process site in the JSR 282 page.

Community Resources

The latest information about the RTSJ is most easily found through the informal RTSJ web site (<http://www.rtsj.org>). This site includes the latest version of the RTSJ in HTML, an archive of earlier versions of the specification, RTSJ example code, papers, useful links (such as a link to the download site for the Reference Implementation), and information about current implementations of the RTSJ.

NIST supports an RTSJ mailing list, *rtj-discuss*, which is an excellent place to exchange email with the RTSJ community. Instructions for joining this mailing list can be found on the informal RTSJ web site (<http://www.rtsj.org>).

Release History

Version 1.0.2

Version 1.0.2 is primarily a set of clarifications and changes derived from exchanges with several teams implementing RTSJ. Version 1.0.2 also tightens the specification a little where those changes will not greatly harm implementability.

Finalization

The revised finalization semantics attempt to clarify the algorithm for finalizing objects in a scope in passes until there are no more finalizable objects or the finalizers create a schedulable object that references the scope. The revised semantics also require the schedulable object exiting the scope when it becomes finalizable to run the finalizers.

Cost enforcement

The concept of *deferred suspension* has been added to cost enforcement for schedulable objects, and *enforced priority* has been defined for processing groups.

AsyncEventHandler

A conflict between the 1.0.1 semantics, and the method documentation and RI was resolved by specifying that an async event handler's fire count is decremented before invoking `handleAsyncEvent`.

The semantics for the fire count manipulation methods have been specified for all callers.

Non-Default Initial Memory Area

Detailed semantics have been added for the treatment of non-default initial memory areas for real-time threads and async event handlers. This had ramifications in the scoped memory `joinAndEnter` methods.

Asynchronously Interrupted Exception

Interruptible blocking methods are defined, as are the semantics for interrupting those methods.

Exceptions

In some cases where there was ambiguity about which exception should be thrown, the exception precedence has been clarified.

Version 1.0.1

The primary objective of the 1.0.1 version of the RTSJ was clarification. That clarification resulted in a nearly complete re-write of the semantics and requirements sections, but those revisions were not intended to express different semantics. They were intended to express the original semantics:

- More carefully and completely
- More conveniently (also more redundantly. Some statements that were made once in the first edition of the RTSJ are now replaced with the method-by-method consequences of those statements.)
- Without constraining the implementation except where that is necessary to achieve compatibility between implementations.

Deprecation in this version should be treated as emphatic advice to avoid the deprecated feature. In RTSJ 1.0.1, deprecation usually means that the semantics for a class or method may not be actively dangerous, but for various reasons its semantics cannot be clarified in a reasonable and unambiguous way. These methods (and one class) are not necessary, and they will almost certainly be entirely removed soon. In any case, their semantics are not well-defined, they cannot be adequately tested, and any application that values portability should not use them.

Requirements

The processing group enforcement option has been separated from the cost enforcement option, and support for processing group deadline less than period has been made optional.

A profile for development tools has been introduced. This permits a development tool to implement RTSJ classes without implementing all the RTSJ semantics.

Threads and Scheduling

As much as possible, semantics that relate to scheduling have been made attributes of the scheduler, this caused many semantics to move from the Threads chapter to the Scheduling chapter.

RealtimeThread

1. Added `MemoryArea getMemoryArea()` to return the initial memory area of the `RealtimeThread`. This method that was in the 1.0 reference implementation and the TCK, but was left out of the specification.
2. Changes to the operation of `setPriority` in `java.lang.Thread` are described.
3. `waitForNextPeriod` is made `static` since it would be dangerous for a thread to invoke the method on any other thread.
4. Added `waitForNextPeriodInterruptible` because (especially for real-time systems) blocking methods should be interruptible.
5. Added two new `setIfFeasible` methods (See the `Schedulable` interface.)

NoHeapRealtimeThread

None

Scheduler

Let `fireSchedulable` throw `UnsupportedOperationException` because only specific classes (possibly no classes) that implement the `Schedulable` interface can be fired by any given scheduler.

Add method

1. `public abstract boolean setIfFeasible(Schedulable schedulable, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)`

The following methods were made abstract:

1. `public abstract boolean setIfFeasible (Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)`
2. `public abstract boolean setIfFeasible (Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)`

Schedulable

A number of methods were consistently present in classes that implement `Schedulable` and should have been included in `Schedulable`. They were added to the `Schedulable` interface:

1. `boolean addIfFeasible()`,

2. `boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory),`
3. `boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group),`
4. `boolean setIfFeasible(ReleaseParameters release, ProcessingGroupParameters group)`

Two methods were added to improve the parallel construction of the Scheduler and the Schedulable interface. The new methods also make it possible to update the scheduling parameters considering feasibility:

1. `boolean setIfFeasible(SchedulingParameters sched, ReleaseParameters release, MemoryParameters memory)`
2. `boolean setIfFeasible(SchedulingParameters sched, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)`

`IllegalThreadStateException` was removed from the throws clause of the `setScheduler` methods because there is no case where that exception will be thrown.

ReleaseParameters

`ReleaseParameters` now implements `Cloneable` and has a public `clone` method.

SchedulingParameters

`SchedulingParameters` now implements `Cloneable` and has a public `clone` method.

The constructor is changed to protected to match similar classes, such as `ReleaseParameters`.

PeriodicParameters

Two new constructors,

1. `PeriodicParameters(RelativeTime period)`
2. `PeriodicParameters(HighResolutionTime start, RelativeTime period)`

were added to conveniently support common patterns, and as pro-active support for a safety-critical specification.

ProcessingGroupParameters

`ProcessingGroupParameters` now implements `Cloneable` and has a public `clone` method.

AperiodicParameters

The methods and constants that relate to the arrival time queue overflow behavior have been moved from `SporadicParameters` to `AperiodicParameters` because the arrival time queue is an aspect of aperiodic tasks. The implementation had to maintain the queue, but the application could not control it unless it used sporadic parameters. The list of moved methods and constants is:

- `String arrivalTimeQueueOverflowExcept`
- `String arrivalTimeQueueOverflowIgnore`
- `String arrivalTimeQueueOverflowReplace`
- `String arrivalTimeQueueOverflowSave`
- `int getInitialArrivalTimeQueueLength()`
- `void setInitialArrivalTimeQueueLength(int length)`
- `void setArrivalTimeQueueOverflowBehavior(String behavior)`
- `String getArrivalTimeQueueOverflowBehavior()`

A new constructor:

`AperiodParameters()`

has been added to make a common case easier to write, and to proactively support work on the safety critical Java specification.

SporadicParameters

The methods and constants listed as moved *to* `AperiodicParameters` are moved *from* `SporadicParameters`. They still appear in the sporadic parameters class, but now they are inherited from aperiodic parameters.

A new constructor:

`SporadicParameters(RelativeTime minInterarrival)`

has been added to make a common case easier to write, and to proactively support work on the safety critical Java specification.

PriorityScheduler

The semantics for the *REPLACE* MIT violation policy for sporadic parameters has been revised to “If the last event has not been processed revise its release time to the current time. This will alter the deadline at any time up to completion of its processing or the time it misses its deadline. If event processing for the replaced release has been completed or it has already missed its deadline, the behavior of the *REPLACE* policy is equivalent to *IGNORE*.”

The `fireSchedulable` method is permitted to throw `UnsupportedOperationException` if the scheduler does not support the method for parameter object. The base instance of the priority scheduler is expected to throw

UnsupportedOperationException in every case. The `fireSchedulable` method has never worked, and its semantics are hard to clarify without significantly extending the semantics of schedulable objects. The method is not deprecated because the semantics of schedulable objects quite likely will be extended to make this method useful, but that goes beyond clarification.

SchedulingParameters

`SchedulingParameters` now implements `Cloneable` and has a public `clone` method.

The constructor is changed from public to protected.

PriorityParameters

No changes

ImportanceParameters

No changes

ProcessingGroupParameters

`ProcessingGroupParameters` now implements `Cloneable` and has a public `clone` method

Deleted and Deprecated Methods

PriorityScheduler

1. `MAX_PRIORITY`, and `MIN_PRIORITY` are deprecated because they may bind the maximum and minimum priorities into an application at compile time and the available priority range is an attribute of the run-time platform.

Memory Management

MemoryArea

The throws clause of:

- `newArray(Class type, int number)`

has been changed from `(IllegalAccessException InstantiationException)` to no exceptions to better reflect the checked exceptions thrown when the underlying platform creates a new array.

The throws clause of:

- `newInstance(reflect.Constructor c, Object[] args)`

has been changed from `(IllegalAccessException InstantiationException)` to `(IllegalAccessException, InstantiationException,`

`InvocationTargetException`) to reflect the checked exceptions thrown when the underlying platform creates a new instance.

The throws clause of:

- `newInstance(Class c)`

has been changed from `(IllegalAccessError InstantiationException)` to `(IllegalAccessError, InstantiationException, NoSuchMethodError)` to reflect the checked exceptions thrown when the underlying platform creates a new instance.

HeapMemory

None

ImmortalMemory

The `executeInArea` method and the family of `newInstance` methods may be used by Java threads. Previously this was ambiguous, but since a Java thread can switch between heap and immortal without the full semantics of a scope stack, this limited access to RTSJ memory areas can be supported without modifying Java threads. Moreover, a mechanism to switch Java threads to “immortal mode” has always been implied by the memory area semantics of static initializers.

SizeEstimator

There was no way to estimate the size of an array object so

```
public void reserveArray(int dimension)
    for arrays of references and
public void reserveArray(int dimension, Class type)
    for arrays of primitive types were added.
```

ScopedMemory

The `getPortal()` method now throws the (unchecked) exceptions `MemoryAccessError` and `IllegalAssignmentError`. These exceptions were implicit in the existing reference and assignment rules, but the exact rules for generating them from this method were unclear. They are declared to make their possibility clear to the caller.

The `ScopedCycleException` exception has been removed from the throws clause of the `joinAndEnter` methods that do not take a time parameter. There is no case where these methods need to throw that exception.

The `joinAndEnter` methods are altered to throw `IllegalArgumentException` immediately when they have no non-null logic value. Previously, this behavior was ambiguous with the possibilities to either return immediately (as stated explicitly) or throw `IllegalArgumentException` as required to behave like indivisible `join` and `enter`.

LMemory

Added four new constructors to simplify a common use case, and for symmetry with the `LPhysicalMemory` class:

1. `LMemory(long size)`
2. `LMemory(long size, Runnable logic)`
3. `LMemory(SizeEstimator size)`
4. `LMemory(SizeEstimator size, Runnable logic)`

VMemory

Added four new constructors to simplify a common use case, and for symmetry with the `VPhysicalMemory` class:

1. `VMemory(long size)`
2. `VMemory(long size, Runnable logic)`
3. `VMemory(SizeEstimator size)`
4. `VMemory(SizeEstimator size, Runnable logic)`

PhysicalMemoryManager

The type of the static final fields `ALIGNED`, `BYTESWAP`, `SHARED`, and `DMA` is changed from `String` to `Object`. This is a compatible change that supports more flexibility of implementation.

A new static final field, `IO_PAGE`, is added.

The methods

- `public static void onInsertion(long base, long size, AsyncEvent ae)`
- `public static void onRemoval(long base, long size, AsyncEvent ae)`
- `public static boolean unregisterInsertionEvent(long base, long size, AsyncEvent ae)`
- `public static boolean unregisterRemovalEvent(long base, long size, AsyncEvent ae)`

have been added to replace deprecated insertion and removal methods.

All methods now throw `OffsetOutOfBoundsException` when the base is negative, and `SizeOutOfBoundsException` when the extent of memory passes the physical or virtual addressing boundary. This brings them in line with the methods in the classes commonly used by applications (such as `ImmutablePhysicalMemory`.)

PhysicalMemoryTypeFilter

The methods

- `public void onInsertion(long base, long size, AsyncEvent ae)`
- `public void onRemoval(long base, long size, AsyncEvent ae)`
- `public boolean unregisterInsertionEvent(long base, long size, AsyncEvent ae)`
- `public boolean unregisterRemovalEvent(long base, long size, AsyncEvent ae)`

have been added to replace deprecated insertion and removal methods.

All methods now throw `OffsetOutOfBoundsException` when the base is negative, and `SizeOutOfBoundsException` when the extent of memory passes the physical or virtual addressing boundary. This brings them in line with the methods in the classes commonly used by applications (such as `ImmutablePhysicalMemory`.)

RawMemoryAccess

The constructors for this class now specifically mention the possibility of `OutOfMemoryError`.

RawMemoryFloatAccess

The constructors for this class now specifically mention the possibility of `OutOfMemoryError`.

LTPhysicalMemory

None

VTPhysicalMemory

None

ImmutablePhysicalMemory

None

MemoryParameters

Changed `setAllocationRateIfFeasible()` to accept a `long` argument for consistency with all the other allocation rate methods that take and return `long`.

The class now implements `Cloneable` and includes `public Object clone()`

GarbageCollector

None

Deprecated Methods

The public constructor for the `GarbageCollector` is deprecated.

The `onInsertion(long base, long size, AsyncEventHandler aeh)` and `onRemoval(long base, long size, AsyncEventHandler aeh)` methods in both

`PhysicalMemoryManager` and `PhysicalMemoryTypeFilter` are deprecated in favor of new methods. The deprecated methods use async event handlers in a unnecessarily clumsy way (without an async event), and causing special argument values to unregister handlers is not good Java practice.

Synchronization

Significant changes have been made to the way priority ceiling protocol interacts with priority inheritance protocol.

Priority ceiling emulation was essentially unworkable as specified in the 1.0 spec. The 1.0.1 revision has semantics that can be implemented, but several changes to the APIs have been necessary.

MonitorControl

Made the constructor protected since `MonitorControl` is abstract, and each subclass should be a singleton.

Changed both `setMonitorControl()` methods to return the old policy instead of returning void.

PriorityCeilingEmulation

Added static `PriorityCeilingEmulation instance(int ceiling)` to return a priority ceiling emulation object for each ceiling value. This lets the implementation check for equal ceilings by checking for equality of the object references.

Added `getCeiling()` replacing `getDefaultCeiling`. The `getCeiling` method has the same semantics as `getDefaultCeiling`. The name, `getDefaultCeiling` is misleading because there is no value that can correctly be called the default ceiling.

Added static `PriorityCeilingEmulation getMaxCeiling()` which returns a singleton universally usable priority ceiling emulation object. This object is usable in cases where an application wishes to make priority ceiling emulation the default monitor control policy.

PriorityInheritance

None

WaitFreeWriteQueue

There is little purpose for the reader and writer parameters for the constructor. The class can support multiple readers and writers, so these are at best hints. The constructor with the reader and writer parameters was retained, but two new constructors without those parameters were added to reduce the confusion caused by those parameters.

Added the constructor `WaitFreeWriteQueue(int maximum)` throws `IllegalArgumentException`

Added the constructor `WaitFreeWriteQueue(int maximum, MemoryArea memory)` throws `IllegalArgumentException`.

Added `InterruptedException` to the throws clause of `read()`.

WaitFreeReadQueue

There is little purpose for the reader and writer parameters for the constructor. The class can support multiple readers and writers, so these are at best hints. The constructor with the reader and writer parameters was retained, but two new constructors without those parameters were added to reduce the confusion caused by those parameters.

Added the constructor `WaitFreeReadQueue(int maximum, boolean notify)` throws `IllegalArgumentException`

Added the constructor `WaitFreeReadQueue(int maximum, MemoryArea memory, boolean notify)` throws `IllegalArgumentException`

Changed the return type of `write(Object object)` from `boolean` to `void` because the method could never return anything but `true`, and added `InterruptedException` to its throws clause because it is a general principle that all blocking methods should be interruptible.

Added `InterruptedException` to the throws clause of `waitForData()` because it is a blocking method that should be interruptible.

WaitFreeDequeue

This class has been deprecated because it does not do anything that the separate `WaitFreeReadQueue` and `WaitFreeWriteQueue` do not do as well, and proper use of the proper read and write methods is unnecessarily confusing.

Changed the return type of `blockingWrite(Object object)` from `boolean` to `void` because the method could never return anything but `true`.

Deleted and Deprecated Methods

The public constructors from the `PriorityCeilingEmulation` and `PriorityInheritance` classes have been removed. An implementation that exposes constructors for these methods as specified in version 1.0 would be needlessly complicated and it must leak immortal memory. The revised APIs require the implementation to produce (possibly lazily) singleton instances for each distinct value of the monitor control classes

PriorityCeilingEmulation

Removed the public constructor because this class is supposed to be able to generate a unique instance per ceiling value.

Deprecated `getDefaultCeiling()` because the method name is misleading. The new `getCeiling` method should be used instead.

PriorityInheritance

Removed the public constructor because this is supposed to be a singleton.

Time

HighResolutionTime

Revised `RelativeTime` `relative(Clock clock, HighResolutionTime time)` to require a `RelativeTime` argument. Previously any `HighResolutionTime` argument was syntactically correct, but only `RelativeTime` could be used without causing the method to throw a runtime exception.

There was no way to recover the clock property of a `HighResolutionTime` object, so the `getClock()` method was added. There are many ways to alter (re-associate) the clock, so no symmetrical `setClock()` method was required.

The signature of `set(HighResolutionTime time)` is not changed, but its meaning is altered. Version 1.0 had it defined to set the value of the parameter to a value corresponding to the current date. This is completely at odds with Java conventions, and while the Javadoc in the reference implementation agrees with the 1.0 specification, the code alters the time value of `this` to match the parameter. The TCK was consistent with the RI code. The most likely conclusion is that the semantics for the method were a cut-and-paste error. The error is so surprising and potentially destructive that instead of deprecating the method (as would be normal for a case like this) the semantics were corrected to agree with the RI and TCK and set the millis and nanos values of `this` to the values from the parameter.

`HighResolutionTime` now implements `Cloneable`. It also has a public `clone` method and public `hashCode` and `equals` methods that work correctly with `clone`.

AbsoluteTime

Added four new constructors that include the clock association:

1. `AbsoluteTime(Clock clock)`
2. `AbsoluteTime(long millis, int nanos, Clock clock)`
3. `AbsoluteTime(AbsoluteTime time, Clock clock)`
4. `AbsoluteTime(java.util.Date date, Clock clock)`

Changed several methods that were specified as `final` to non-`final`. Some arithmetic methods were `final` and some were not with no discernible rationale for the difference. Changing them all to non-`final` was the most compatible way to resolve the inconsistency.

- `add(RelativeTime time)`,
- `subtract(AbsoluteTime time)`
- `subtract(AbsoluteTime time, RelativeTime destination)`

- `subtract(RelativeTime time)`

were specified as `final` and now have that attribute removed.

One version of the `relative` method:

```
RelativeTime relative (Clock clock, AbsoluteTime dest)
```

could not be implemented as specified since it called for returning a `RelativeTime` value in an `AbsoluteTime` object. It was changed to:

```
RelativeTime relative (Clock clock, RelativeTime dest)
```

RelativeTime

Added three new constructors that include the clock association:

1. `RelativeTime(Clock clock)`
2. `RelativeTime(long millis, int nanos, Clock clock)`
3. `RelativeTime(RelativeTime time, Clock clock)`

Changed two methods that were specified as `final` to non-`final`. Some arithmetic methods were `final` and some were not with no discernible rationale for the difference. Changing them all not non-`final` was the most compatible way to resolve the inconsistency.

- `add(RelativeTime time)`
- `subtract(RelativeTime time)`

were specified as `final` and now have that attribute removed.

RationalTime

None. The class is deprecated.

Deprecated Methods

HighResolutionTime

none.

RelativeTime

- `getInterarrivalTime()`,
- `getInterarrivalTime(RelativeTime destination)`, and
- `addInterarrivalTo(AbsoluteTime timeAndDestination)`

are deprecated because their only purpose is to support the deprecated `RationalTime` class.

Deprecated Classes

RationalTime

The `RationalTime` class was defined so it has two reasonable and incompatible descriptions. Neither of them can be implemented well. We have deprecated the class and hope to revisit the underlying concepts and abstract them better in a future revision.

Clocks and Timers

Clock

Added `RelativeTime` `getEpochOffset()` throws `UnsupportedOperationException` to let applications compare clocks with different epochs, and to let them discover clocks that have no concept of an epoch.

Changed the return type of `getTime(AbsoluteTime time)` from `void` to `AbsoluteTime` to make the method consistent with the RTSJ conventions of returning a reference to the destination parameter when it is provided. This is also required to give it consistent behavior when a null parameter is passed.

Timer

The `fire()` method inherited from `AsyncEventHandler` is now overridden in this class since this is the most appropriate place to note that it throws `UnsupportedOperationException` if the class is `Timer`.

The void `start(boolean disabled)` throws `IllegalStateException` method has been added because without such a method there would be no way to start a timer in the disabled state that does not have a possible race condition.

Added new method:

1. `public AbsoluteTime getFireTime(AbsoluteTime fireTime)`

OneShotTimer

No changes except those inherited from `Timer`.

PeriodicTimer

No changes except those inherited from `Timer`.

Asynchrony

AsyncEventHandler

Two methods were defined as `final` with no justification when compared to other methods in the same class that were not `final`:

1. `getAndClearPendingFireCount`
2. `getPendingFireCount`

Their `final` attribute was removed to make them consistent with other similar methods.

Two new methods were added:

1. `boolean isDaemon()`
2. `void setDaemon(boolean on)`

To control the “daemon nature” of the AEH.

The two methods added to the `Schedulable` interface are also added here.

AsyncEvent

None.

BoundAsyncEventHandler

The `BoundAsyncEventHandler` class is no longer an abstract class. Since the class includes a constructor with a `logic` parameter, it could operate without being subclassed. There was no reason it should be abstract, and leaving it abstract was inconvenient for application developers.

Interruptible

None

AsynchronouslyInterruptedException

Added the `boolean clear()` method to implement the safe semantics of `happened()`, but not offer to secretly throw AIE.

Timed

None.

Deprecated Methods

AsynchronouslyInterruptedException

Deprecated `happened()` and `propagate()`. These methods are defined to throw the `AsynchronouslyInterruptedException` and they do not include that *checked* exception in their throws clauses. They may be actively dangerous to methods up their call chain that are not expecting an exception, but the danger is not bad enough to justify deleting these methods without warning.

An application can achieve the effect of `propagate` by throwing an AIE after it catches it, and it can achieve the effect of `happened` by combining the new `clear()` method with `fire`.

System and Options

POSIXSignalHandler

Deprecated many signal names that are not found in the POSIX 9945-1-1996 standard:

- SIGCANCEL,
- SIGFREEZE,
- SIGIO,
- SIGLOST,
- SIGWP,
- SIGPOLL,
- SIGPROF,
- SIGPWR,
- SIGTHAW,
- SIGURG,
- SIGVTALRM,
- SIGWAITING,
- SIGWINCH,
- SIGXCPU, and
- SIGXFSZ.

Removed the default no-arg constructor leaving the class with no public constructor.

RealtimeSecurity

Added methods

```
public void checkSetMonitorControl(MonitorControl policy) throws  
SecurityException
```

and

```
public void checkSetDaemon() throws SecurityException
```

because the specification already says that these operations are checked by the security manager.

RealtimeSystem

The no-arg constructor was an artifact of javadoc. Since this class's implementation is entirely static, the constructor is pointless and has been removed.

If applications execute the method call,
`System.getProperty("java.xml.version")`, the return value will be a string of the form, "x.y.z". Where 'x' is the major version number and 'y' and 'z' are

minor version numbers. These version numbers state to which version of the RTSJ the underlying implementation claims conformance. The first release of the RTSJ, dated 11/2001, is numbered as, 1.0.0. Since this property is required in only subsequent releases of the RTSJ implementations of the RTSJ which intend to conform to 1.0.0 may return the String “1.0.0” or null.

Added `getInitialMonitorControl()` to support the monitor control classes.

Exceptions

Added the class `ArrivalTimeQueueOverflowException` to indicate overflow of an async event handlers arrival queue, and `CeilingViolationException` to signify that a thread has attempted to lock a priority ceiling lock when its base priority exceeds the priority of the lock.

The following exceptions have been changed from checked to unchecked:

- `MemoryScopeException`
- `InnaccessibleAreaException`
- `MemoryTypeConflictException`
- `MITViolationException`
- `OffsetOutOfBoundsException`
- `SizeOutOfBoundsException`
- `UnsupportedPhysicalMemoryException`

Each of these exceptions is characteristic of a programming error, not a fault that a programmer should anticipate and handle.

Preface to the First Edition

Dreams

In 1997 the idea of writing real-time applications in the Java programming language seemed unrealistic. Real-time programmers talk about wanting consistent timing behavior more than absolute speed, but that doesn't mean they don't require excellent overall performance. The Java runtime is sometimes interpreted, and almost always uses a garbage collector. The early versions were not known for their blistering performance.

Nevertheless, Java platforms were already being incorporated into real-time systems. It is fairly easy to build a hybrid system that uses C for modules that have real-time requirements and other components written to the Java platform. It is also possible to implement the Java interpreter in hardware (for performance), and integrate the system without a garbage collector (for consistent performance). aJfile Systems produces a Java processor with acceptable real-time characteristics.

Until the summer of 1998, efforts toward support for real-time programming on the Java platform were fragmented. Kelvin Nilsen from NewMonics and Lisa Carnahan from the National Institute for Standards and Technology (NIST) led one effort, Greg Bollella from IBM led a group of companies that had a stake in Java technology and real-time, and Sun had an internal real-time project based on the Java platform.

In the summer of 1998 the three groups merged. The real-time requirements working group included Kelvin Nilsen from NewMonics, Bill Foote and Kevin Russell from Sun, and the group of companies led by Greg Bollella. It also included a diverse selection of technical people from across the real-time industry and a few representatives with a more marketing or management orientation.

The requirements group convened periodically until early 1999. Its final output was a document, *Requirements for Real-time Extensions for the Java Platform*, detailing the requirements the group had developed, and giving some rationale for those requirements. It can be found on the web at <http://www.nist.gov/rt-java> (<http://www.nist.gov/rt-java>).

Realization

One of the critical events during this process occurred in late 1998, when Sun created the *Java Community Process*. Anyone who feels that the Java platform needs a new facility can formally request the enhancement. If the request, called a Java

Specification Request (JSR), is accepted, a *call for experts* is posted. The *specification lead* is chosen and then he or she forms the *expert group*. The result of the effort is a specification, reference implementation, and test suite.

In late 1998, IBM asked Sun to accept a JSR, *The Real-Time Specification for Java*, based partly on the work of the Requirements Working Group. Sun accepted the request as JSR-000001. Greg Bollella was selected as the specification lead. He formed the expert group in two tiers. The primary group:

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

would actually write the specification, and the consultant group:

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

would serve as a pool of readily available expertise and as initial reviewers of early drafts.

The effort commenced in March 1999 with a plenary meeting of the consultant and primary groups at the Chicago Hilton and Towers. This was an educational meeting where the consultants each presented selections of general real-time wisdom, and the specific requirements of their part of the real-time world.

The basis of the specification was laid down at the first primary group meeting. It took place in one of the few civilized locations in the United States that is not accessible to digital or analog cell phone traffic, Mendocino, California. This is also, in the expert opinion of the primary group, the location of a restaurant that produces the world's most heavily cheesed pizza.

Through 1999 the primary group met slightly more than once a month, and meetings for the joint primary and consultants groups were held slightly less than once a month. We worked hard and had glorious fun. Mainly, the fun was the joy of solving a welter of problems with a team of diverse and talented software architects, but there were memorable nontechnical moments.

There was the seminal “under your butt” insight, when James told Greg that he should stop looking over his head for the sense of an argument: “This is simple, Greg. It’s not over your head, it’s going under your butt.” That was the same Burlington, Massachusetts, meeting where a contingent of the expert group attended the 3:00 AM second showing of the newly released *Star Wars Phantom Menace*. The only sane reason for waking up at a time more suitable for going to sleep was that James had gone back to California to attend the movie with his wife, who had purchased tickets weeks in advance. It tickled our fancy to use the magic of time zones and early rising to see the new release before them.

The cinnamon rolls in Des Moines, which David later claimed were bigger than his head. This was an exaggeration. Each roll was slightly less than half the size of David’s head.

The “dead cat” meeting in Ottawa, where Greg claimed that when he took his earache to the clinic, the doctor would probably remove a dead cat.

The “impolite phrase” meeting, also in Ottawa. The group made it into a computer industry gossip column, and our feelings on the thrill of being treated like movie stars simply cannot be expressed in this book. We are, however, impressed that a writer old enough to perceive Greg as IBM’s *boy* is still writing regularly.

In September 1999, the draft specification was published for formal review by participants in the Java Community Process and informal reading by anyone who downloaded it from the group’s web site (<http://www.rtj.org> (<http://www.rtj.org>)). In December 1999, the revised and extended document was published on the web site for public review. Public review remained open until the 14th of February 2000 (yes,

Valentine's Day). Then the specification was revised a final time to address the comments from the general public.

The first result of this work is the document you are reading. IBM is also producing a reference implementation and a test suite to accompany this specification.

Acknowledgments

The reader should consider this work truly a collaborative effort. Many people contributed in diverse ways. Unlike most traditional published books this work is the result of effort and contribution from engineers, executives, administrators, marketing and product managers, industry consultants, and university faculty members spread across more than two dozen companies and organizations from around the globe. It is also the result of a new and unique method for developing software, The Java Community Process.

We'll start at the beginning. Many of the technical contributors came together at a series of forums conceived and hosted by Lisa Carnahan at the National Institute for Standards and Technology. One of the authors, Greg Bollella, was instrumental, along with Lisa, in the early establishment of the organization of the future authors. He thanks his managers at IBM, Ruth Taylor, Rod Smith, and Pat Sultz, for (in their words) being low-maintenance managers and for allowing Greg the freedom to pursue his goal.

The Java Community Process was developed at Sun Microsystems by Jim Mitchell, Ken Urquhart, and others to allow and promote the broad involvement of the computer industry in the development of the Java™ platform. We thank them and all those at Sun and other companies who reviewed the initial proposals of the process. Vicki Shipkowitz the embedded Java product manager at Sun has also helped the Real-Time for Java Expert Group with logistics concerning demonstrations and presentations of the RTSJ.

The Real-Time for Java Expert Group comprises an engineering team and a consultant team. The authors of this work are the primary engineers and we sincerely thank the consultants, mentioned by name previously, for their efforts during the early design phase and for reviewing various drafts. Along the way Ray Kamin, Wolfgang Pieb, and Edward Wentworth replaced three of the original consultants and we thank them for their effort as well.

We thank all those, but especially Kirk Reinholtz of NASA's Jet Propulsion Lab, who submitted comments during the participant and public reviews.

We thank Lisa Friendly, the Java Series editor at Sun Microsystems, and Mike Hendrickson, Sarah Weaver, and Julie DiNicola at Addison-Wesley for their effort in the preparation of this book.

We all thank Russ Richards at DISA for his support of our effort.

We thank Kevin Russell and Bill Foote of Sun Microsystems who worked hard during the NIST sponsored requirements phase.

Although they have much left to do and will likely give us more work as they implement the RTSJ, we thank the reference implementation team at IBM. Peter Haggart leads the team of David Wendt and Jim Mickelson. Greg also thanks them for their efforts on the various robot demonstrations he used in his talks about the RTSJ.

Greg would like to personally thank his dissertation advisor Kevin Jeffay for his guidance.

We thank Robin Coron and Feng Liu, administrative assistants at Sun Microsystems and IBM, respectively, for their logistical support.

A Note on Format

We used javadoc on Java source files to produce most of this book and thus many references to class, interface, and method names use the `@link` construct to produce a hyperlink in the (more typical) html formatted output. Of course, clicking on the hyperlink in the html formatted version will display the definition of the class. We tried to preserve this hyperlink characteristic when this book is formatted for PDF by including on each occurrence of a name the page number of its definition as a trailing subscript.

Unofficial

Introduction

The Real-Time for Java Expert Group (RTJEG), convened under the Java Community Process and JSR-000001, was given the responsibility of producing a specification for extending *The Java Language Specification* and *The Java Virtual Machine Specification* and of providing an Application Programming Interface that will enable the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints (also known as real-time threads). This introduction describes the guiding principles that the RTJEG created and used during their work, a description of the real-time Java requirements developed under the auspices of The National Institute for Standards and Technology (NIST), and a brief, high-level description of each of the seven areas identified as requiring enhancements to accomplish the Expert Group's goals.

Guiding Principles

The guiding principles are high-level statements that delimit the scope of the work of the RTJEG and introduce compatibility requirements for *The Real-Time Specification for Java*.

Applicability to Particular Java Environments: The RTSJ shall not include specifications that restrict its use to particular Java environments, such as a particular version of the Java Development Kit, the Embedded Java Application Environment, or the Java 2 Micro Edition™.

Backward Compatibility: The RTSJ shall not prevent existing, properly written, non-real-time Java programs from executing on implementations of the RTSJ.

Write Once, Run Anywhere: The RTSJ should recognize the importance of “Write Once, Run Anywhere”, but it should also recognize the difficulty of achieving WORA for real-time programs and not attempt to increase or maintain binary portability at the expense of predictability.

Current Practice vs. Advanced Features: The RTSJ should address current real-time system practice as well as allow future implementations to include advanced features.

Predictable Execution: The RTSJ shall hold predictable execution as first priority in all trade-offs; this may sometimes be at the expense of typical general-purpose computing performance measures.

No Syntactic Extension: In order to facilitate the job of tool developers, and thus to increase the likelihood of timely implementations, the RTSJ shall not introduce new keywords or make other syntactic extensions to the Java language.

Allow Variation in Implementation Decisions: The RTJEG recognizes that implementations of the RTSJ may vary in a number of implementation decisions, such as the use of efficient or inefficient algorithms, trade-offs between time and space efficiency, inclusion of scheduling algorithms not required in the minimum implementation, and variation in code path length for the execution of byte codes. The RTSJ should not mandate algorithms or specific time constants for such, but require that the semantics of the implementation be met. The RTSJ offers implementers the flexibility to create implementations suited to meet the requirements of their customers.

Overview of the Seven Enhanced Areas

In each of the seven sections that follow we give a brief statement of direction for each area. These directions were defined at the first meeting of the eight primary engineers in Mendocino, California, in late March 1999, and further clarified through late September 1999.

Thread Scheduling and Dispatching: In light of the significant diversity in scheduling and dispatching models and the recognition that each model has wide applicability in the diverse real-time systems industry, we concluded that our direction for a scheduling specification would be to allow an underlying scheduling mechanism to be used by real-time Java threads but that we would not specify in advance the nature of all (or even a number of) possible scheduling mechanisms. The specification is constructed to allow implementations to provide unanticipated scheduling algorithms. Implementations will allow the programmatic assignment of parameters appropriate for the underlying scheduling mechanism as well as providing any necessary methods for the creation, management, admittance, and termination of real-time Java threads. We also expect that, for now, particular thread scheduling and dispatching mechanisms are bound to an implementation. However, we provide

enough flexibility in the thread scheduling framework to allow future versions of the specification to build on this release and allow the dynamic loading of scheduling policy modules.

To accommodate current practice the RTSJ requires a base scheduler in all implementations. The required base scheduler will be familiar to real-time system programmers. It is priority-based, preemptive, and must have at least 28 unique priorities.

Memory Management: We recognize that automatic memory management is a particularly important feature of the Java programming environment, and we sought a direction that would allow, as much as possible, the job of memory management to be implemented automatically by the underlying system and not intrude on the programming task. Additionally, we understand that many automatic memory management algorithms, also known as garbage collection (GC), exist, and many of those apply to certain classes of real-time programming styles and systems. In our attempt to accommodate a diverse set of GC algorithms, we sought to define a memory allocation and reclamation specification that would:

- be independent of any particular GC algorithm,
- allow the program to precisely characterize a implemented GC algorithm's effect on the execution time, preemption, and dispatching of real-time Java threads, and
- allow the allocation and reclamation of objects outside of any interference by any GC algorithm.

Synchronization and Resource Sharing: Logic often requires serial access to resources. Real-time systems introduce an additional complexity: controlling priority inversion. We have decided that the least intrusive specification for allowing real-time safe synchronization is to require that implementations of the Java keyword `synchronized` include one or more algorithms that prevent priority inversion among real-time Java threads that share the serialized resource. We also note that in some cases the use of the `synchronized` keyword implementing the required priority inversion algorithm is not sufficient to both prevent priority inversion and allow a thread to have an execution eligibility logically higher than the garbage collector. We provide a set of wait-free queue classes to be used in such situations.

Asynchronous Event Handling: Real-time systems typically interact closely with the real-world. With respect to the execution of logic, the real-world is asynchronous. We thus felt compelled to include efficient mechanisms for programming disciplines that would accommodate this inherent asynchrony. The RTSJ generalizes the Java language's mechanism of asynchronous event handling. Required classes represent things that can happen and logic that executes when those things happen. A notable feature is that the execution of the logic is scheduled and dispatched by an implemented scheduler.

Asynchronous Transfer of Control: Sometimes the real-world changes so drastically (and asynchronously) that the current point of logic execution should be immediately and efficiently transferred to another location. The RTSJ includes a mechanism which extends Java's exception handling to allow applications to programatically change the locus of control of another Java thread. It is important to note that the RTSJ restricts this asynchronous transfer of control to logic specifically written with the assumption that its locus of control may asynchronously change.

Asynchronous Thread Termination: Again, due to the sometimes drastic and asynchronous changes in the real-world, application logic may need to arrange for a real-time Java thread to expeditiously and safely transfer its control to its outermost scope and thus end in a normal manner. Note that unlike the traditional, unsafe, and deprecated Java mechanism for stopping threads, the RTSJ's mechanism for asynchronous event handling and transfer of control is safe.

Physical Memory Access: Although not directly a real-time issue, physical memory access is desirable for many of the applications that could productively make use of an implementation of the RTSJ. We thus define a class that allows programmers byte-level access to physical memory as well as a class that allows the construction of objects in physical memory.

Design

The RTSJ comprises seven areas of extended semantics. This chapter introduces the extensions. Further detail, exact requirements, and rationale are given in the opening section of each relevant chapter. The seven areas are discussed in approximate order of their relevance to real-time programming. However, the semantics and mechanisms of each of the areas—scheduling, memory management, synchronization, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, and physical memory access—are all crucial to the acceptance of the RTSJ as a viable real-time development platform.

Scheduling

One of the concerns of real-time programming is to ensure the timely or predictable execution of sequences of machine instructions. Various scheduling schemes name these sequences of instructions differently. Typically used names include threads, tasks, modules, and blocks. The RTSJ introduces the concept of a *schedulable object*. These are the objects that the base scheduler manages, `RealtimeThread` and its subclasses and `AsyncEventHandler` and its subclasses.

Timely execution of schedulable objects means that the programmer can determine by analysis of the program, testing the program on particular implementations, or both whether particular threads will always complete execution before a given timeliness constraint. This is the essence of real-time programming: the addition of temporal constraints to the correctness conditions for computation. For example, for a program to compute the sum of two numbers it may no longer be

acceptable to compute only the correct arithmetic answer but the answer must be computed before a particular time. Typically, temporal constraints are deadlines expressed in either relative or absolute time.

We use the term *scheduling* (or *scheduling algorithm*) to refer to the production of a sequence (or ordering) for the execution of a set of schedulable objects (a *schedule*). This schedule attempts to optimize a particular metric (a metric that measures how well the system is meeting the temporal constraints). A *feasibility analysis* determines if a schedule has an acceptable value for the metric. For example, in hard real-time systems the typical metric is “number of missed deadlines” and the only acceptable value for that metric is zero. So-called soft real-time systems use other metrics (such as mean tardiness) and may accept various values for the metric in use.

Many systems use thread priority in an attempt to determine a schedule. Priority is typically an integer associated with a schedulable object; these integers convey to the system the order in which the threads should execute. The generalization of the concept of priority is *execution eligibility*. We use the term *dispatching* to refer to that portion of the system which selects the thread with the highest execution eligibility from the pool of threads that are ready to run. In current real-time system practice, the assignment of priorities is typically under programmer control as opposed to under system control. The RTSJ’s base scheduler also leaves the assignment of priorities under programmer control. However, the base scheduler also inherits methods from its superclass that may help determine feasibility.

For the base scheduler the feasibility methods may assume a sufficiently fast processor to complete any proposed load on schedule. The RTSJ expects that the base scheduler may be subclassed in particular implementations (e.g., an EDF scheduler) and for those implementations the feasibility methods may correctly indicate the actual feasibility of the system under the given scheduler. Note that for the base scheduler the RTSJ is no different than most real-time operating systems in current use.

The RTSJ requires a number of classes with names of the format `<string>Parameters` (such as `SchedulingParameters`). An instance of one of these parameter classes holds a particular resource-demand characteristic for one or more schedulable objects. For example, the `PriorityParameters` subclass of `SchedulingParameters` contains the execution eligibility metric of the base scheduler, i.e., priority. At some time (construction-time or later when the parameters are replaced using setter methods), instances of parameter classes are bound to a schedulable object. The schedulable object then assumes the characteristics of the values in the parameter object. For example, if a `PriorityParameters` instance that had in its `priority` field the value representing the highest priority available is bound to a schedulable object, then that object will assume the characteristic that it will execute whenever it is ready in preference to all other schedulable objects (except, of course, those also with the highest priority).

The RTSJ is written so as to allow implementers the flexibility to install arbitrary scheduling algorithms and feasibility analysis algorithms in an implementation of the specification. We do this because the RTJEG understands that the real-time systems industry has widely varying requirements with respect to scheduling. Use of the Java platform may help produce code written once but able to execute on many different computing platforms (known as Write Once, Run Anywhere.) The RTSJ both contributes to this goal and detracts from it. The RTSJ's rigorous specification of the required priority scheduler is critical for portability of time-critical code, but the RTSJ permits and supports platform-specific schedulers which are not portable.

Memory Management

Garbage-collected memory heaps have always been considered an obstacle to real-time programming due to the unpredictable latencies introduced by the garbage collector. The RTSJ addresses this issue by providing several extensions to the memory model, which support memory management in a manner that does not interfere with the ability of real-time code to provide deterministic behavior. This goal is accomplished by allowing the allocation of objects outside of the garbage-collected heap for both short-lived and long-lived objects.

Memory Areas

The RTSJ introduces the concept of a memory area. A memory area represents an area of memory that may be used for the allocation of objects. Some memory areas exist outside of the heap and place restrictions on what the system and garbage collector may do with objects allocated within. Objects in some memory areas are never garbage collected; however, the garbage collector must be capable of scanning these memory areas for references to any object within the heap to preserve the integrity of the heap.

There are four basic types of memory areas:

1. Scoped memory provides a mechanism, more general than stack allocated objects, for managing objects that have a lifetime defined by scope.
2. Physical memory allows objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access.
3. Immortal memory represents an area of memory containing objects that may be referenced without exception or garbage collection delay by any schedulable object, specifically including no-heap real-time threads and no-heap asynchronous event handlers.
4. Heap memory represents an area of memory that is the heap. The RTSJ does not change the determinant of lifetime of objects on the heap. The lifetime is still

determined by visibility.

Scoped Memory

The RTSJ introduces the concept of scoped memory. A memory scope is used to give bounds to the lifetime of any objects allocated within it. When a scope is entered, every use of `new` causes the memory to be allocated from the active memory scope. A scope may be entered explicitly, or it can be attached to a schedulable object which will effectively enter the scope before it executes the object's `run()` method.

The contents of a scoped memory are discarded when no object in the scope can be referenced. This is done by a technique similar to reference counting the scope. A conformant implementation might maintain a count of the number of external references to each memory area. The reference count for a `ScopedMemory` area would be increased by entering a new scope through the `enter()` method of `MemoryArea`, by the creation of a schedulable object using the particular `ScopedMemory` area, or by the opening of an inner scope. The reference count for a `ScopedMemory` area would be decreased when returning from the `enter()` method, when the schedulable object using the `ScopedMemory` terminates, or when an inner scope returns from its `enter()` method. When the count drops to zero, the `finalize` method for each object in the memory would be executed to completion. Reuse of the scope is blocked until finalization is complete.

Scopes may be nested. When a nested scope is entered, all subsequent allocations are taken from the memory associated with the new scope. When the nested scope is exited, the previous scope is restored and subsequent allocations are again taken from that scope.

Because of the lifetimes of scoped objects, it is necessary to limit the references to scoped objects, by means of a restricted set of assignment rules. A reference to a scoped object cannot be assigned to a variable from an outer scope, or to a field of an object in either the heap or the immortal area. A reference to a scoped object may only be assigned into the same scope or into an inner scope. The virtual machine must detect illegal assignment attempts and must throw an appropriate exception when they occur.

The flexibility provided in choice of scoped memory types allows the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.

Immortal Memory

`ImmortalMemory` is a memory resource shared among all schedulable objects and threads in an application. Objects allocated in `ImmortalMemory` are always available to non-heap threads and asynchronous event handlers without the possibility of a delay for garbage collection.

Budgeted Allocation

The RTSJ also provides limited support for providing memory allocation budgets for schedulable objects using memory areas. Maximum memory area consumption and maximum allocation rates for individual schedulable objects may be specified when they are created.

Synchronization

Terms

For the purposes of this section, the use of the term *priority* should be interpreted somewhat more loosely than in conventional usage. In particular, the term *highest priority thread* merely indicates the most eligible thread—the thread that the dispatcher would choose among all of the threads that are ready to run—and doesn't necessarily presume a strict priority based dispatch mechanism.

Wait Queues

Threads and asynchronous event handlers waiting to acquire a resource must be released in execution eligibility order. This applies to the processor as well as to synchronized blocks. If schedulable objects with the same execution eligibility are possible under the active scheduling policy, such schedulable objects are awakened in FIFO order. For example:

- Threads waiting to enter synchronized blocks are granted access to the synchronized block in execution eligibility order.
- A blocked thread that becomes ready to run is given access to the processor in execution eligibility order.
- A thread whose execution eligibility is explicitly set by itself or another thread is given access to the processor in execution eligibility order.
- A thread that performs a yield will be given access to the processor after waiting threads of the same execution eligibility.
- Threads that are preempted in favor of a thread with higher execution eligibility may be given access to the processor at any time as determined by a particular implementation. The implementation is required to provide documentation stating exactly the algorithm used for granting such access.

Priority Inversion Avoidance

Any conforming implementation must provide an implementation of the synchronized primitive with default behavior that ensures that there is no unbounded priority inversion. Furthermore, this must apply to code if it is run within the implementation as well as to real-time threads. The priority inheritance protocol must

be implemented by default. The priority inheritance protocol is a well-known algorithm in the real-time scheduling literature and it has the following effect. If thread t_1 attempts to acquire a lock that is held by a lower-priority thread t_2 , then t_2 's priority is raised to that of t_1 as long as t_2 holds the lock (and recursively if t_2 is itself waiting to acquire a lock held by an even lower-priority thread).

The specification also provides a mechanism by which the programmer can override the default system-wide policy, or control the policy to be used for a particular monitor, provided that policy is supported by the implementation. The monitor control policy specification is extensible so that new mechanisms can be added by future implementations.

A second policy, priority ceiling emulation protocol (or highest locker protocol), is also specified for systems that support it. This protocol is also a well-known algorithm in the literature; somewhat simplified, its effect is as follows:

- A monitor is given a “priority ceiling” when it is created; the programmer should choose the highest priority of any thread that could attempt to enter the monitor.
- As soon as a thread enters synchronized code, its (active) priority is raised to the monitor's ceiling priority. If, through programming error, a thread has a higher base priority than the ceiling of the monitor it is attempting to enter, then an exception is thrown.
- On leaving the monitor, the thread has its active priority reset. In simple cases it will set be to the thread's previous active priority, but under some circumstances (e.g. a dynamic change to the thread's base priority while it was in the monitor) a different value is possible

Note that while the RTSJ requires that the execution of non-heap schedulable objects must not be delayed by garbage collection on behalf of lower-priority schedulable objects, an application can cause a no-heap schedulable object to wait for garbage collection by synchronizing using an object between an heap-using thread or schedulable object and a non-heap schedulable object. The RTSJ provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and no-heap real-time threads. These classes are provided explicitly to enable communication between the real-time execution of non-heap schedulable objects and regular Java threads or heap-using schedulable objects.

Determinism

Conforming implementations shall provide a fixed upper bound on the time required to enter a synchronized block for an unlocked monitor.

Asynchronous Event Handling

The asynchronous event facility comprises two classes: `AsyncEvent` and `AsyncEventHandler`. An `AsyncEvent` object represents something that can happen, like a POSIX signal, a hardware interrupt, or a computed event like an airplane entering a specified region. When one of these events occurs, which is indicated by the `fire()` method being called, the associated instances of `AsyncEventHandler` are scheduled and the `handleAsyncEvent()` methods are invoked, thus the required logic is performed. Also, methods on `AsyncEvent` are provided to manage the set of instances of `AsyncEventHandler` associated with the instance of `AsyncEvent`.

An instance of `AsyncEventHandler` can be thought of as something similar to a thread. It is a `Runnable` object: when the event fires, the associated handlers are scheduled and the `handleAsyncEvent()` methods are invoked. What distinguishes an `AsyncEventHandler` from a simple `Runnable` is that an `AsyncEventHandler` has associated instances of `ReleaseParameters`, `SchedulingParameters` and `MemoryParameters` that control the actual execution of the handler once the associated `AsyncEvent` is fired. When an event is fired, the handlers are executed asynchronously, scheduled according to the associated `ReleaseParameters` and `SchedulingParameters` objects, in a manner that looks like the handler has just been assigned to its own thread. It is intended that the system can cope well with situations where there are large numbers of instances of `AsyncEvent` and `AsyncEventHandler` (tens of thousands). The number of fired (in process) handlers is expected to be smaller.

A specialized form of an `AsyncEvent` is the `Timer` class, which represents an event whose occurrence is driven by time. There are two forms of `Timers`: the `OneShotTimer` and the `PeriodicTimer`. Instances of `OneShotTimer` fire once, at the specified time. Periodic timers fire initially at the specified time, and then periodically according to a specified interval.

Timers are driven by `Clock` objects. There is a special `Clock` object, `Clock.getRealtimeClock()`, that represents the real-time clock. The `Clock` class may be extended to represent other clocks the underlying system might make available (such as a execution time clock of some granularity).

Asynchronous Transfer of Control

Many times a real-time programmer is faced with a situation where the computational cost of an algorithm is highly variable, the algorithm is iterative, and the algorithm produces successively refined results during each iteration. If the system, before commencing the computation, can determine only a time bound on how long to execute the computation (i.e., the cost of each iteration is highly variable and the minimum required latency to terminate the computation and receive the last consistent result is much less than about half of the mean iteration cost), then asynchronously

transferring control from the computation to the result transmission code at the expiration of the known time bound is a convenient programming style. The RTSJ supports this and other styles of programming where such transfer is convenient with a feature termed Asynchronous Transfer of Control (ATC).

The RTSJ's approach to ATC is based on several guiding principles, informally outlined in the following lists.

Methodological Principles

- A method must explicitly indicate its susceptibility to ATC. Since legacy code or library methods might have been written assuming no ATC, by default ATC must be turned off (more precisely, must be deferred as long as control is in such code).
- Even if a method allows ATC, some code sections must be executed to completion and thus ATC is deferred in such sections. These ATC-deferred sections are synchronized methods, static initializers, and synchronized statements.
- Code that responds to an ATC does not return to the point in the schedulable object where the ATC was triggered; that is, an ATC is an unconditional transfer of control. Resumptive semantics, which returns control from the handler to the point of interruption, are not needed since they can be achieved through other mechanisms (in particular, an `AsyncEventHandler`).

Expressibility Principles

- A mechanism is needed through which an ATC can be explicitly triggered in a target schedulable object. This triggering may be direct (from a source thread or schedulable object) or indirect (through an asynchronous event handler).
- It must be possible to trigger an ATC based on any asynchronous event including an external happening or an explicit event firing from another thread or schedulable object. In particular, it must be possible to base an ATC on a timer going off.
- Through ATC it must be possible to abort a real-time thread but in a manner that does not carry the dangers of the `Thread` class's `stop()` and `destroy()` methods.

Semantic Principles

- If ATC is modeled by exception handling, there must be some way to ensure that an asynchronous exception is only caught by the intended handler and not, for example, by an all-purpose handler that happens to be on the propagation path.
- Nested ATCs must work properly. For example, consider two, nested ATC-based timers and assume that the outer timer has a shorter time-out than the nested, inner timer. If the outer timer times out while control is in the nested code of the inner timer, then the nested code must be aborted (as soon as it is outside an ATC-deferred section), and control must then transfer to the appropriate catch clause

for the outer timer. An implementation that either handles the outer time-out in the nested code, or that waits for the longer (nested) timer, is incorrect.

Pragmatic Principles

- There should be straightforward idioms for common cases such as timer handlers and real-time thread termination.
- If code with a time-out completes before the timer's expiration, the timer needs to be automatically stopped and its resources returned to the system.

Asynchronous Real-Time Thread Termination

Although not a only real-time issue, many event-driven computer systems that tightly interact with external real-world non-computer systems (e.g., humans, machines, control processes, etc.) may require mode changes in their computational behavior as a result of significant changes in the non-computer real-world system. It would be convenient to program threads that abnormally terminate when the external real-time system changes in a way such that the thread is no longer useful. Without this facility, a thread or set of threads have to be coded in such a manner so that their computational behavior anticipated all of the possible transitions among possible states of the external system. It is an easier design task to code threads to computationally cooperate for only one (or a very few) possible states of the external system. When the external system makes a state transition, the changes in computation behavior might then be managed by an oracle, that terminates a set of threads useful for the old state of the external system, and invokes a new set of threads appropriate for the new state of the external system. Since the possible state transitions of the external system are encoded in only the oracle and not in each thread, the overall system design is easier.

Earlier versions of the Java language supplied mechanisms for achieving these effects: in particular the methods `stop()` and `destroy()` in class `Thread`. However, since `stop()` could leave shared objects in an inconsistent state, `stop()` has been deprecated. The use of `destroy()` can lead to deadlock (if a thread is destroyed while it is holding a lock) and although it was not deprecated until version 1.5 of the Java specification, its usage has long been discouraged. A goal of the RTSJ was to meet the requirements of asynchronous thread termination without introducing the dangers of the `stop()` or `destroy()` methods.

The RTSJ accommodates safe asynchronous real-time thread termination through a combination of the asynchronous event handling and the asynchronous transfer of control mechanisms. To create such a set of real-time threads consider the following steps:

- Make all of the application methods of the real-time thread interruptible

- Create an oracle which monitors the external world by binding a number of asynchronous event handlers to happenings which occur at appropriate mode changes
- Have the handlers call `interrupt()` on each of the real-time threads affected by the change
- After the handlers call `interrupt()` have them create a new set of real-time threads appropriate to the current state of the external world

The effect of the happening is then to cause each interruptible method to abort abnormally by transferring control to the appropriate catch clause. Ultimately the `run()` method of the real-time thread will complete normally.

This idiom provides a quick (if coded to be so) but orderly clean up and termination of the real-time thread. Note that the oracle can comprise as many or as few asynchronous event handlers as appropriate.

Physical Memory Access

The RTSJ defines classes for programmers wishing to directly access physical memory from code written in the Java language. `RawMemoryAccess` defines methods that allow the programmer to construct an object that represents a range of physical addresses. Access to the physical memory is then accomplished through `get[type]()` and `set[type]()` methods of that object where the type represents a word size, i.e., byte, short, int, long, float, and double. No semantics other than the `set[type]()` and `get[type]()` methods are implied. The `VTPhysicalMemory`, `LTPhysicalMemory`, and `ImmortalPhysicalMemory` classes allow programmers to construct an object that represents a range of physical memory addresses. When this object is used as a `MemoryArea` other objects can be constructed in the physical memory using the `new` keyword as appropriate.

The `PhysicalMemoryManager` is available for use by the various physical memory accessor objects (`VTPhysicalMemory`, `LTPhysicalMemory`, `ImmortalPhysicalMemory`, `RawMemoryAccess`, and `RawMemoryFloatAccess`) to create objects of the correct type that are bound to areas of physical memory with the appropriate characteristics - or with appropriate accessor behavior. Examples of characteristics that might be specified are: DMA memory, accessors with byte swapping, etc. OEMs may provide `PhysicalMemoryTypeFilter` classes that allow additional characteristics of memory devices to be specified.

Raw Memory Access

An instance of `RawMemoryAccess` models a range of physical memory as a fixed sequence of bytes. A full complement of accessor methods allow the contents of the physical area to be accessed through offsets from the base, interpreted as byte, short, int, or long data values or as arrays of these types.

Whether the offset specifies the most-significant or least-significant byte of a multibyte value is affected by the `BYTE_ORDER` static variable in class `RealtimeSystem`, possibly amended by a byte swapping attribute associated with the underlying physical memory type.

The `RawMemoryAccess` class allows a real-time program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software.

A raw memory area cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking) and error prone (since it is sensitive to the specific representational choices made by the Java compiler).

Physical Memory Areas

In many cases, systems needing the predictable execution of the RTSJ will also need to access various kinds of memory at particular addresses for performance or other reasons. Consider a system in which very fast static RAM was programmatically available. A design that could optimize performance might wish to place various frequently used Java objects in the fast static RAM. The `VTPhysicalMemory`, `LTPhysicalMemory`, and `ImmortalPhysicalMemory` classes allow the programmer this flexibility. The programmer would construct a physical memory object on the memory addresses occupied by the fast RAM.

Exceptions

The RTSJ introduces several new exceptions, and some new treatment of exceptions surrounding asynchronous transfer of control and memory allocators.

Unofficial

Requirements and Conventions

The base requirements of this specification are:

1. Except as specifically required by this specification, any implementation shall fully conform to a Java platform configuration.
2. Except as noted in this chapter, any implementation of this specification shall implement all classes and methods in this specification. In particular, every implementation must include a conformant implementation of the `PriorityScheduler` class.
3. The `javax.realtime` package shall contain no public or protected methods not included in this specification.
4. The JVM shall not be implemented in a way that permits unbounded priority inversion in any scheduling interaction it implements.
5. Subject to the usual assumptions, the methods in `javax.realtime` can safely be used concurrently by multiple threads unless it is otherwise documented.
6. No specific instance of `PhysicalMemoryTypeFilter` is required, but every implementation must support at least one such instance with the characteristic that it supports access to the range of physical memory that can be made accessible to the implementation.
7. Static final values, as found in `AperiodicParameters`, `PhysicalMemoryManager`, `SporadicParameters`, `RealtimeSystem`, and `PriorityScheduler`,

must be handled by the implementation such that their values cannot be resolved at compile time.

Many aspects of this specification set a minimum requirement, but permit the implementation latitude in its implementation. For instance, the required priority scheduler requires at least 28 consecutively numbered real-time priorities. It does not, however, specify the numeric value of the maximum and minimum priorities. Implementations are encouraged to offer as many real-time priority levels as they can support.

Except where otherwise specified, when this specification requires object creation the object is created in the current allocation context.

Optional Facilities

There are no bounds on extensions based on this specification, except that only extensions that conform with future versions of this specification may be implemented in the `javax.realtime` package.

Several optional extensions are included in this specification. An application cannot depend on these facilities in every implementation, but if an optional facility is implemented, the application may rely on it to behave as specified here. Those extensions are:

Cost enforcement	Allows the application to control the processor utilization of a schedulable object.
Processing Group enforcement	Allows the application to control the processor utilization of a group of schedulable objects
Processing Group deadline less than period	Allows the application to specify a processing group deadline less than the processing group period
Priority Ceiling Emulation Protocol	An alternative to priority inheritance for priority inversion avoidance
Atomic access to raw memory	Most atomic access is optional. The implementation may provide the raw memory access characteristics in system properties of the form <code>javax.realtime.atomicaccess_<xxx></code> .
Allocation-rate enforcement on heap allocation	Allows the application to limit the rate at which a schedulable object creates objects in the heap.

The `ProcessingGroupParameters` class is only functional on systems that support the processing group enforcement option. Cost enforcement, and cost overruns handlers are only functional on systems that support the cost enforcement option. If processing group enforcement is supported, `ProcessingGroupParameters` must function as specified. If cost enforcement is supported, cost enforcement, and cost overrun handlers must function as specified.

If the processing group deadline less than period is not supported, values passed to the constructor for `ProcessingGroupParameters` and its `setDeadline` method are constrained to be equal to the period. If the option is supported, processing group deadlines less than the period must be supported and function as specified.

If priority ceiling emulation is supported, `PriorityCeilingEmulation` must be implemented as specified. If priority ceiling emulation is not supported, `PriorityCeilingEmulation` must be present, but the implementation may not permit its use as a monitor control policy.

If heap allocation rate enforcement is supported, it must be implemented as specified. If heap allocation rate enforcement is not supported, the allocation rate attribute of `MemoryParameters` must be checked for validity but otherwise ignored by the implementation.

The following semantics are optional for an RTSJ implementation designed and licensed exclusively as a development tool:

- The priority scheduler need not support fixed-priority preemptive scheduling or priority inheritance. This does not excuse an implementation from fully supporting the relevant APIs. It only reduces the required behavior of the underlying scheduler to the level of the scheduler in the Java specification extended to at least 28 priorities.
- No semantics constraining timing beyond the requirements of the Java specifications need be supported. Specifically, garbage collection may delay any thread without bound and any delay in delivering asynchronously interrupted exceptions is permissible including never delivering the exception. Note, however, that if any AIE other than the generic AIE is delivered, it must meet the AIE semantics, and all heap-memory-related semantics other than preemption remain fully in effect. Further, relaxed timing does not imply relaxed sequencing. For instance, semantics for scoped memory must be fully implemented.
- The RTSJ semantics that alter standard Java method behavior—such as the modified semantics for `Thread.setPriority` and `Thread.interrupt`—are not required for a development tool, but such deviations from the RTSJ must be documented, and the implementation must be able to generate a run-time warning each time one of these methods deviates from standard RTSJ behavior.
- These relaxed requirements set a floor for RTSJ development system tool imple-

mentations. A development tool may choose to implement semantics that are not required.

Conditionally-Required Facilities

An implementation must support conditionally-required facilities if the underlying hardware and software permits. This specification includes three conditionally-required facilities:

POSIXSignalHandler	This class shall be implemented on every platform where POSIX signals are supported
RawMemoryFloatAccess	This shall be implemented on every platform for which the base JVM includes support for the float and double types.
Mapping memory	If the system supports address translation, the implementation shall support the memory mapping features of the raw memory access classes.

If POSIX signals are not supported, the `POSIXSignalHandler` class must not be present. If POSIX signals are supported, `POSIXSignalHandler` must be implemented as specified.

If floating point is not supported by the platform, `RawMemoryFloatAccess` must not be present. If floating point is supported, then `RawMemoryFloatAccess` must be implemented as specified.

Required Documentation

Each implementation of the RTSJ is required to provide documentation for several behaviors:

1. If the feasibility testing algorithm is not the default, document the feasibility testing algorithm.
2. If schedulers other than the base priority scheduler are available to applications, document the behavior of the scheduler and its interaction with each other scheduler as detailed in the Scheduling chapter. Document the list of classes that constitute schedulable objects for the scheduler unless that list is the same as the list of schedulable objects for the base scheduler. If there are restrictions on use of the scheduler from a non-heap context, document those restrictions.
3. A schedulable object that is preempted by a higher-priority schedulable object is placed in the queue for its active priority, at a position determined by the imple-

mentation. If the preempted schedulable object is not placed at the front of the appropriate queue the implementation must document the algorithm used for such placement. Placement at the front of the queue may be required in a future version of this specification.

4. If the implementation supports cost enforcement, then the implementation is required to document the granularity at which the current CPU consumption is updated.
5. The memory mapping implemented by any physical memory type filter must be documented unless it is a simple sequential mapping of contiguous bytes.
6. The implementation must fully document the behavior of any subclasses of `GarbageCollector`.
7. An implementation that provides any `MonitorControl` subclasses not detailed in this specification must document their effects, particularly with respect to priority inversion control and which (if any) schedulers fail to support the new policy.
8. If on losing “boosted” priority due to a priority inversion avoidance algorithm, the schedulable object is not placed at the front of its new queue, the implementation must document the queuing behavior.
9. For any available scheduler other than the base scheduler an implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default `PriorityInheritance` monitor control policy. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the base priority scheduler found in the Synchronization chapter. If there are restrictions on use of the scheduler from a no-heap context, the documentation must detail the effect of these restrictions for each RTSJ API.
10. The worst-case response interval between firing an `AsyncEvent` because of a bound happening to releasing an associated `AsyncEventHandler` (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture.
11. The interval between firing an `AsynchronouslyInterruptedException` on an ATC-enabled thread and first delivery of that exception (assuming no higher-priority schedulable objects are runnable) must be documented for some reference architecture.
12. If cost enforcement is supported, and the implementation assigns the cost of running finalizers for objects in scoped memory to any schedulable object other than the one that caused the scope’s reference count to drop to zero by leaving the scope, the rules for assigning the cost shall be documented.
13. If cost enforcement is supported, and enforcement (blocked-by-cost-override) can

be delayed beyond the enforcement time granularity, the maximum such delay shall be documented.

14. If the implementation of `RealtimeSecurity` is more restrictive than the required implementation, or has run-time configuration options, those features shall be documented.
15. For each supported clock, the documentation must specify whether the resolution is settable, and if it is settable the documentation must indicate the supported values.
16. If an implementation includes any clocks other than the required real-time clock, their documentation must indicate in what contexts those clocks can be used. If they cannot be used in no-heap context, the documentation must detail the consequences of passing the clock, or a time that uses the clock to a no-heap schedulable object.

Conventions

Throughout the RTSJ, when we use the word *code*, we mean code written in the Java programming language. When we mention the Java language in the RTSJ, that also refers to the Java programming language. The use of the term *heap* in the RTSJ will refer to the heap used by the runtime of the Java language.

Definitions

A *thread* is an instance of the `java.lang.Thread` class.

A *real-time thread* is an instance of the `javax.realtime.RealtimeThread` class.

A *Java thread* is a thread that is not a real-time thread.

A *no-heap real-time thread* is an instance of the `javax.realtime.NoHeapRealtimeThread` class.

An *asynchronous event handler* is an instance of the `javax.realtime.AsyncEventHandler` class.

The term `Schedulable` object is distinct from the term *schedulable object* (SO). Every object that implements the `Schedulable` interface can be termed a `Schedulable` object, but only objects that are recognized as dispatchable entities by the base scheduler are *schedulable objects* with respect to that scheduler. The base scheduler's set of schedulable objects comprises instances of `RealtimeThread` and `AsyncEventHandler`. Other schedulers may support a different set of schedulable objects, but this specification only defines the behavior of the base scheduler so the term *schedulable object* should be understood as “schedulable by the base scheduler.”

Standard Java Classes

In several cases the semantics of the RTSJ influence the semantics of classes from the standard Java class libraries. Specifically:

- The set and get methods for priority in `java.lang.Thread` for real-time threads.
- The `ThreadGroup` class' behavior with respect to real-time threads.
- The behavior of the `ThreadGroup`-related methods in `Thread` when they are applied to real-time threads.

Priority

The methods `setPriority` and `getPriority` in `java.lang.Thread` are `final`. The real-time thread classes are consequently not able to override them and modify their behavior to suit the requirements of the RTSJ scheduler. To bring the `java.lang.Thread` class in line with its real-time sub-classes, the semantics of the `getPriority` and `setPriority` methods are modified as follows:

- `Thread.setPriority()`:
 - a. Use of `Thread.setPriority()` must not affect the correctness of the priority inversion avoidance algorithms controlled by `PriorityCeilingEmulation` and `PriorityInheritance`. Changes to the base priority of a real-time thread as a result of invoking `Thread.setPriority()` are governed by semantics from *Synchronization*.
 - b. Real-time threads may use `setPriority` to access the expanded range of priorities available to real-time threads. If the real-time thread's priority param-

ters object is not shared, `setPriority` behaves effectively as if it included the code snippet:

```
PriorityParameters pp = getSchedulingParameters();
pp.setPriority(newPriority);
```

- c. If the real-time thread's priority parameters object is shared with other schedulable objects, `setPriority` must give the thread an unshared `PriorityParameters` instance allocated in the same memory area as the real-time thread object and containing the new priority value.
- d. `setPriority` throws `IllegalArgumentException` if the thread is a real-time thread and the new priority is outside the range allowed by the real-time thread's scheduler.
- e. `setPriority` throws `ClassCastException` if the thread is a real-time thread and its scheduling parameters object is not an instance of `PriorityParameters`.
- `Thread.getPriority()`:
 - a. When used on a real-time thread, `getPriority` behaves effectively as if it included the code snippet:


```
((PriorityParameters)t.getSchedulingParameters()).getPriority();
```
 - b. If the scheduling parameters are not of type `PriorityParameters`, then a `ClassCastException` is thrown.

All supported monitor control policies must apply to Java threads as well as to all schedulable objects.

Thread Groups

Thread groups are rooted at a base `ThreadGroup` object which may be created in heap or immortal memory. All thread group objects hold references to all their member threads, and subgroups, and a reference to their parent group. Since heap and immortal memory can not hold references to scoped memory, it follows that a thread group can never be allocated in scoped memory. It then follows that no thread allocated in scoped memory may be referenced from any thread group, and consequently such threads are not part of any thread group and will hold a null thread group reference. Similarly, a `NoHeapRealtimeThread` can not be a member of a heap allocated thread group.

1. Real-time threads with null thread groups are not included when thread groups are enumerated, interrupted, stopped, resumed, or suspended. However, when the current thread is a real-time thread with a null thread group:
 - a. The `Thread.enumerate` class method returns the integer 1, and populates its

- array argument with the current real-time thread.
- b. `Thread.activeCount` returns 1.
 - c. `Thread.getThreadGroup` returns null in all cases, not only when the thread has terminated.
2. A Java thread that is created from a real-time thread inherits the thread group of the real-time thread, if it has one; otherwise an attempt is made to add it to the application root thread group. The constructor shall throw a `SecurityException` if the Java thread is not permitted to use the application root thread group.
 3. The thread group of a Java thread created by an async event handler is assigned as if it was created by a real-time thread without a thread group (as described in 2.)
 4. A thread group cannot be created in scoped memory. The constructor shall throw an `IllegalAssignmentError`.
 5. Limits on priority set in the thread group have no influence on real-time threads.
 6. Except as specified previously, real-time threads have the same `ThreadGroup` membership rules as the parent `Thread` class.

InterruptedException

The interruptible methods in the standard libraries (such as `Object.wait`, `Thread.sleep`, and `Thread.join`) have their contract expanded slightly such that they will respond to interruption not only when the `interrupt` method is invoked on the current thread, but also, for schedulable objects, when executing within a call to `AIE.doInterruptible` and that `AIE` is fired. See `Asynchrony`.

System Properties

System properties and their `String` values allocated during system initialization shall be allocated in immortal memory.

Unofficial

Real-Time Threads

This section describes the two real-time thread classes. These classes:

- Provide for the creation of real-time threads that have more precise scheduling semantics than `java.lang.Thread`.
- Provide for the creation of real-time threads that have no dependency on the heap.

The `RealtimeThread` class extends `java.lang.Thread`. The `ReleaseParameters`, `SchedulingParameters`, and `MemoryParameters` objects passed to the `RealtimeThread` constructor allow the temporal and processor demands of the thread to be communicated to the scheduler.

The `NoHeapRealtimeThread` class extends `RealtimeThread`. A `NoHeapRealtimeThread` is not allowed to allocate or even reference objects from the Java heap, and can thus safely execute in preference to the garbage collector.

Overview

The RTSJ provides two types of objects which implement the `Schedulable` interface: real-time threads and asynchronous event handlers. This chapter defines the facilities that are available to real-time threads. In many cases these facilities are also available to asynchronous event handlers. In particular:

- the default scheduler must support the scheduling of both real-time threads and asynchronous event handlers;
- real-time threads and asynchronous event handlers are allowed to enter into mem-

ory areas and consequently they have associated scope stacks;

- the flow of control of real-time threads and asynchronous event handlers are affected by the RTSJ asynchronous transfer of control facilities;

Where the semantics and requirements apply to both real-time threads and asynchronous event handlers, the term schedulable object will be used.

Semantics and Requirements for Real-time Threads

1. Garbage collection executing in the context of a Java thread must not in itself block execution of a no-heap thread with a higher execution eligibility, however application locks work as specified even when the lock causes synchronization between a heap-using thread and a no-heap thread.
2. Each real-time thread has an attribute which indicates whether an `AsynchronouslyInterruptedException` is pending. This attribute is set when a call to `RealtimeThread.interrupt()` is made on the associated real-time thread, and when an asynchronously interrupted exception's `fire` method is invoked between the time the real-time thread has entered that exception's `doInterruptible` method, and return from `doInterruptible`. (See the *Asynchrony* chapter.)
3. A call to `RealtimeThread.interrupt()` generates the system's generic `AsynchronouslyInterruptedException`. (See the *Asynchrony* chapter.)
4. The `RealtimeThread.waitForNextPeriod` and `waitForNextPeriodInterruptible` methods are for use by real-time threads that have periodic release parameters. In the absence of any deadline miss or cost overrun (or an interrupt in the case of `waitForNextPeriodInterruptible`) the methods return when the real-time thread's next period is due.
5. In the presence of a cost overrun or a deadline miss, the behavior of `waitForNextPeriod` is governed by the thread's scheduler.
6. Instances of `RealtimeThread` that are created in scoped memory and instances of `NoHeapRealtimeThread` do not have conventional references to thread groups nor do thread groups have conventional references to these threads. For the purposes of this version of the specification those references are `null`.
7. Real-time threads with null thread groups handle uncaught exceptions as if the thread used the `uncaughtException` method in `ThreadGroup`:
 - if the exception is a subclass of `ThreadDeath` the thread simply terminates
 - otherwise the thread prints a stack trace of the exception to `System.err`

before it terminates.

Rationale

The Java platform's priority-preemptive dispatching model is very similar to the dispatching model found in the majority of commercial real-time operating systems. However, the dispatching semantics were purposefully relaxed in order to allow execution on a wide variety of operating systems. Thus, it is appropriate to specify real-time threads by extending `java.lang.Thread`. The `ReleaseParameters` and `MemoryParameters` provided to the `RealtimeThread` constructor allow for a number of common real-time thread types, including periodic threads.

The `NoHeapRealtimeThread` class is provided in order to allow time-critical threads to execute in preference to the garbage collector given appropriate assignment of execution eligibility. The memory access and assignment semantics of the `NoHeapRealtimeThread` are designed to guarantee that the execution of such threads does not lead to an inconsistent heap state.

5.1 RealtimeThread

Declaration

```
public class RealtimeThread extends java.lang.Thread implements
    Schedulableg1
```

All Implemented Interfaces: `java.lang.Runnable`, `Schedulableg1`

Direct Known Subclasses: `NoHeapRealtimeThreadg5`

Description

Class `RealtimeThread` extends `java.lang.Thread` and adds access to real-time services such as asynchronous transfer of control, non-heap memory, and advanced scheduler services.

As with `java.lang.Thread`, there are two ways to create a usable `RealtimeThread`.

- Create a new class that extends `RealtimeThread` and override the `run()` method with the logic for the thread.
- Create an instance of `RealtimeThread` using one of the constructors with a `logic` parameter. Pass a `Runnable` object whose `run()` method implements the logic of the thread.

5.1.1 Constructors

```
public RealtimeThread()
```

Create a real-time thread with default values for all parameters. This constructor is equivalent to `RealtimeThread(null, null, null, null, null, null)`.

```
public RealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling)
```

Create a real-time thread with the given `SchedulingParameters112` and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, null, null, null, null, null)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the scheduling parameters are not compatible with the associated scheduler.

`IllegalAssignmentError448` - Thrown if the new `RealtimeThread` instance cannot hold a reference to `scheduling`, or if `scheduling` cannot hold a reference to the new `RealtimeThread`.

```
public RealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release)
```

Create a real-time thread with the given `SchedulingParameters112` and `ReleaseParameters116` and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, release, null, null, null, null)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the scheduling parameters or release parameters are not compatible with the associated scheduler.

`IllegalAssignmentError448` - Thrown if the new `RealtimeThread` instance cannot hold a reference to `scheduling` or `release`, or if either parameter cannot hold a reference to the new `RealtimeThread`.

```
public RealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.MemoryArea161 area,
    javax.realtime.ProcessingGroupParameters143 group,
    java.lang.Runnable logic)
```

Create a real-time thread with the given characteristics and a specified `java.lang.Runnable`. The thread group of the new thread is inherited from its creator unless the newly-created real-time thread is allocated in scoped memory, then its thread group is (effectively) null.

The newly-created real-time thread is associated with the scheduler in effect during execution of the constructor.

Parameters:

`scheduling` - The `SchedulingParameters112` associated with `this` (And possibly other instances of `Schedulable81`). If `scheduling` is null and the creator is a schedulable object, `SchedulingParameters112` is a clone of the creator's value created in the same memory area as `this`. If `scheduling` is null and the creator is a Java thread, the contents and type of the new `SchedulingParameters` object is governed by the associated scheduler.

`release` - The `ReleaseParameters116` associated with `this` (and possibly other instances of `Schedulable81`). If `release` is null the new `RealtimeThread` will use a clone of the default `ReleaseParameters` for the associated scheduler created in the memory area that contains the `RealtimeThread` object.

`memory` - The `MemoryParameters273` associated with `this` (and possibly other instances of `Schedulable81`). If `memory` is null, the new `RealtimeThread` receives null value for its memory parameters, and the amount or rate of memory allocation for the new thread is unrestricted.

`area` - The `MemoryArea161` associated with `this`. If `area` is null, the initial memory area of the new `RealtimeThread` is the current memory area at the time the constructor is called.

`group` - The `ProcessingGroupParameters143` associated with `this` (and possibly other instances of `Schedulable81`). If null, the new `RealtimeThread` will not be associated with any processing group.

`logic` - The `Runnable` object whose `run()` method will serve as the logic for the new `RealtimeThread`. If `logic` is null, the `run()` method in the new object will serve as its logic.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the parameters are not compatible with the associated scheduler.

`IllegalAssignmentError448` - Thrown if the new `RealtimeThread` instance cannot hold a reference to non-null values of `scheduling`, `release memory` and `group`, or if those parameters cannot hold a reference to the new `RealtimeThread`. Also thrown if the new `RealtimeThread` instance cannot hold a reference to non-null values of `area` or `logic`.

5.1.2 Methods

`public boolean addIfFeasible()`

Description copied from interface: `javax.realtime.Schedulable81`

This method first performs a feasibility analysis with `this` added to the system. If the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of `Schedulable81` should be considered in feasibility analysis until further notified. If the analysis showed that the system including `this` would not be feasible, this method does not admit `this` to the feasibility set.

If the object is already included in the feasibility set, do nothing.

Specified By: `addIfFeasible81` in interface `Schedulable81`

Returns: True if inclusion of `this` in the feasibility set yields a feasible system, and false otherwise. If true is returned then `this` is known to be in the feasibility set. If false is returned `this` was not added to the feasibility set, but may already have been present.

`public boolean addToFeasibility()`

Description copied from interface: `javax.realtime.Schedulable81`

Inform the scheduler and cooperating facilities that this instance of `Schedulable81` should be considered in feasibility analysis until further notified.

If the object is already included in the feasibility set, do nothing.

Specified By: [addToFeasibility₈₂](#) in interface [Schedulable₈₁](#)

Returns: True, if the resulting system is feasible. False, if not.

```
public static javax.realtime.RealtimeThread29
    currentRealtimeThread()
```

Gets a reference to the current instance of `RealtimeThread`.

It is permissible to call `currentRealtimeThread` when control is in an [AsyncEventHandler₃₉₃](#). The method will return a reference to the `RealtimeThread` supporting that release of the async event handler.

Returns: A reference to the current instance of `RealtimeThread`.

Throws:

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

```
public void deschedulePeriodic()
```

If the [ReleaseParameters₁₁₆](#) object associated with this `RealtimeThread` is an instance of [PeriodicParameters₁₂₂](#), perform any *deschedulePeriodic* actions specified by this thread's scheduler. If the type of the associated instance of [ReleaseParameters₁₁₆](#) is not [PeriodicParameters₁₂₂](#) nothing happens.

```
public static javax.realtime.MemoryArea161
    getCurrentMemoryArea()
```

Return a reference to the [MemoryArea₁₆₁](#) object representing the current allocation context.

If this method is invoked from a Java thread it will return that thread's current memory area (heap or immortal.)

Returns: A reference to the [MemoryArea₁₆₁](#) object representing the current allocation context.

```
public static int getInitialMemoryAreaIndex()
```

Returns the position in the initial memory area stack, of the initial memory area for the current real-time thread. Memory area stacks may include inherited stacks from parent threads. The initial memory area of a `RealtimeThread` or `AsyncEventHandler` is the memory area given as a parameter to its constructor. The index in the initial memory area stack of the initial memory area is a fixed property of the real-time thread.

If the current memory area stack of the current real-time thread is not the original stack and the memory area at the initial memory area index is not the initial memory area, then `IllegalStateException` is thrown.

Returns: The index into the initial memory area stack of the initial memory area of the current `RealtimeThread`.

Throws:

`java.lang.IllegalStateException` - Thrown if the memory area at the initial memory area index, in the current scope stack is not the initial memory area.

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

```
public javax.realtime.MemoryArea161 getMemoryArea()
```

Return the initial memory area for this `RealtimeThread` (corresponding to the area parameter for the constructor.)

Note: Unlike the scheduling-related parameter objects, there is never a case where a default parameter will be constructed for the thread. The default is a *reference* to the current allocation context when `this` is constructed.

Returns: A reference to the initial memory area for this thread.

Since: 1.0.1

```
public static int getMemoryAreaStackDepth()
```

Gets the size of the stack of `MemoryArea161` instances to which the current schedulable object has access.

Note: The current memory area (`getCurrentMemoryArea()33`) is found at memory area stack index `getMemoryAreaStackDepth() - 1`.

Returns: The size of the stack of `MemoryArea161` instances.

Throws:

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

```
public javax.realtime.MemoryParameters273
    getMemoryParameters()
```

Description copied from interface: `javax.realtime.Schedulable81`

Gets a reference to the `MemoryParameters273` object for this schedulable object.

Specified By: [getMemoryParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [MemoryParameters₂₇₃](#) object.

```
public static javax.realtime.MemoryArea161
    getOuterMemoryArea(int index)
```

Gets the instance of [MemoryArea₁₆₁](#) in the memory area stack at the index given. If the given index does not exist in the memory area scope stack then null is returned.

Note: The current memory area ([getCurrentMemoryArea\(\)₃₃](#)) is found at memory area stack index [getMemoryAreaStackDepth\(\)](#) - 1., so [getCurrentMemoryArea\(\)](#) == [getOutMemoryArea\(getMemoryAreaStackDepth\(\) - 1\)](#).

Parameters:

`index` - The offset into the memory area stack.

Returns: The instance of [MemoryArea₁₆₁](#) at index or null if the given value is does not correspond to a position in the stack.

Throws:

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

```
public javax.realtime.ProcessingGroupParameters143
    getProcessingGroupParameters()
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [ProcessingGroupParameters₁₄₃](#) object for this schedulable object.

Specified By: [getProcessingGroupParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [ProcessingGroupParameters₁₄₃](#) object.

```
public javax.realtime.ReleaseParameters116
    getReleaseParameters()
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [ReleaseParameters₁₁₆](#) object for this schedulable object.

Specified By: [getReleaseParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [ReleaseParameters₁₁₆](#) object.

public [javax.realtime.Scheduler](#)₉₇ [getScheduler](#)()

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

Gets a reference to the [Scheduler](#)₉₇ object for this schedulable object.

Specified By: [getScheduler](#)₈₂ in interface [Schedulable](#)₈₁

Returns: A reference to the associated [Scheduler](#)₉₇ object.

public [javax.realtime.SchedulingParameters](#)₁₁₂

[getSchedulingParameters](#)()

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

Gets a reference to the [SchedulingParameters](#)₁₁₂ object for this schedulable object.

Specified By: [getSchedulingParameters](#)₈₂ in interface [Schedulable](#)₈₁

Returns: A reference to the current [SchedulingParameters](#)₁₁₂ object.

public void [interrupt](#)()

Extends the function of [Thread.interrupt\(\)](#), generates the generic [AsynchronouslyInterruptedException](#) and targets it at this, and sets the interrupted state to pending. (See [AsynchronouslyInterruptedException](#)₄₂₂ .

The semantics of [Thread.interrupt\(\)](#) are preserved.

Overrides: [interrupt](#) in class [Thread](#)

public boolean [removeFromFeasibility](#)()

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

Inform the scheduler and cooperating facilities that this instance of [Schedulable](#)₈₁ should *not* be considered in feasibility analysis until it is further notified.

Specified By: [removeFromFeasibility](#)₈₃ in interface [Schedulable](#)₈₁

Returns: True, if the removal was successful. False, if the schedulable object cannot be removed from the scheduler's feasibility set; e.g., the schedulable object is not part of the scheduler's feasibility set.

public void **schedulePeriodic()**

Begin unblocking [waitForNextPeriod\(\)](#)₅₄ for a periodic thread. If deadline miss detection is disabled, enable it. Typically used when a periodic schedulable object is in a deadline miss condition.

The details of the interaction of this method with [deschedulePeriodic\(\)](#)₃₃ and [waitForNextPeriod\(\)](#)₅₄ are dictated by this thread's scheduler.

If this `RealTimeThread` does not have a type of [PeriodicParameters](#)₁₂₂ as its [ReleaseParameters](#)₁₁₆ nothing happens.

public boolean **setIfFeasible**([javax.realtime.ReleaseParameters](#)₁₁₆ release, [javax.realtime.MemoryParameters](#)₂₇₃ memory)

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. If the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setIfFeasible](#)₈₃ in interface [Schedulable](#)₈₁

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `waitForNextPeriod()54` or `waitForNextPeriodInterruptible()54`.

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: `javax.realtime.Schedulable81`

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setIfFeasible84` in interface `Schedulable81`

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

memory - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

group - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

[java.lang.IllegalThreadStateException](#) - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [waitForNextPeriod\(\)₅₄](#) or [waitForNextPeriodInterruptible\(\)₅₄](#) .

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setIfFeasible`₈₅ in interface `Schedulable`₈₁

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃ .)

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃ .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `waitForNextPeriod()`₅₄ or `waitForNextPeriodInterruptible()`₅₄ .

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: `javax.realtime.Schedulable`₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the

resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setIfFeasible`₈₆ in interface `Schedulable`₈₁

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if `this` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[waitForNextPeriod\(\)](#)₅₄ or
[waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setIfFeasible](#)₈₈ in interface [Schedulable](#)₈₁

Parameters:

- `scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made.
False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `waitForNextPeriod()54` or `waitForNextPeriodInterruptible()54`.

```
public void setMemoryParameters(
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: `javax.realtime.Schedulable81`

Sets the memory parameters associated with this instance of `Schedulable`. This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the memory parameters of the existing schedulable objects, this may change the feasibility of the current system.

Specified By: `setMemoryParameters89` in interface `Schedulable81`

Parameters:

memory - A `MemoryParameters273` object which will become the memory parameters associated with this after the method call. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Throws:

`java.lang.IllegalArgumentException` - Thrown if memory is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and memory is located in heap memory.

`IllegalAssignmentError448` - Thrown if the schedulable object cannot hold a reference to memory, or if memory cannot hold a reference to this schedulable object instance.

```
public boolean setMemoryParametersIfFeasible(
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: `javax.realtime.Schedulable81`

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setMemoryParametersIfFeasible90` in interface `Schedulable81`

Parameters:

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

IllegalAssignmentError₄₄₈ - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

```
public void setProcessingGroupParameters(
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁
Sets the [ProcessingGroupParameters](#)₁₄₃ of this.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the processing group parameters of the existing schedulable objects, this may change the feasibility of the current system.

Specified By: [setProcessingGroupParameters](#)₉₀ in interface [Schedulable](#)₈₁

Parameters:

group - A [ProcessingGroupParameters](#)₁₄₃ object which will take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Throws:

[java.lang.IllegalArgumentException](#) - Thrown when group is not compatible with the scheduler for this schedulable object. Also thrown if this schedulable object is no-heap and group is located in heap memory.

IllegalAssignmentError₄₄₈ - Thrown if this object cannot hold a reference to group or group cannot hold a reference to this.

```
public boolean setProcessingGroupParametersIfFeasible(
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setProcessingGroupParametersIfFeasible₉₁](#) in interface [Schedulable₈₁](#)

Parameters:

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

```
public void setReleaseParameters(
    javax.realtime.ReleaseParameters116 release)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Sets the release parameters associated with this instance of [Schedulable](#).

Since this affects the constraints expressed in the release parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. The different properties of the release parameters may take effect at different times. See the documentation for the scheduler for details.

Specified By: [setReleaseParameters₉₂](#) in interface [Schedulable₈₁](#)

Parameters:

`release` - A [ReleaseParameters₁₁₆](#) object which will become the release parameters associated with this after the method call, and take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `release` is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and `release` is located in heap memory.

`IllegalAssignmentError448` - Thrown if this object cannot hold a reference to `release` or `release` cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [waitForNextPeriod\(\)₅₄](#) or [waitForNextPeriodInterruptible\(\)₅₄](#).

```
public boolean setReleaseParametersIfFeasible(
    javax.realtime.ReleaseParameters116 release)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setReleaseParametersIfFeasible93` in interface `Schedulable81`

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `waitForNextPeriod()54` or `waitForNextPeriodInterruptible()54`.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler)
```

Description copied from interface: `javax.realtime.Schedulable81`
Sets the reference to the Scheduler object. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and `scheduler`.

Specified By: `setScheduler94` in interface `Schedulable81`

Parameters:

`scheduler` - A reference to the scheduler that will manage execution of this schedulable object. Null is not a permissible value.

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduler` is null, or the schedulable object's existing parameter values are not compatible with `scheduler`. Also

thrown if this schedulable object is no-heap and scheduler is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if the schedulable object cannot hold a reference to scheduler.

`java.lang.SecurityException` - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler,
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memoryParameters,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Sets the scheduler and associated parameter objects. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and scheduler.

Specified By: [setScheduler₉₄](#) in interface [Schedulable₈₁](#)

Parameters:

`scheduler` - A reference to the scheduler that will manage the execution of this schedulable object. Null is not a permissible value.

`scheduling` - A reference to the [SchedulingParameters₁₁₂](#) which will be associated with this. If null, the default value is governed by scheduler (a new object is created if the default value is null). (See [PriorityScheduler₁₀₃](#).)

`release` - A reference to the [ReleaseParameters₁₁₆](#) which will be associated with this. If null, the default value is governed by scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memoryParameters` - A reference to the [MemoryParameters₂₇₃](#) which will be associated with this. If null, the default value is governed by scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`group` - A reference to the [ProcessingGroupParameters₁₄₃](#) which will be associated with this. If null, the default value is governed by scheduler (a new object is created). (See [PriorityScheduler₁₀₃](#).)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable object is no-heap and `scheduler`, `scheduling release`, `memoryParameters`, or `group` is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if this object cannot hold references to all the parameter objects or the parameters cannot hold references to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `waitForNextPeriod()`₅₄ or `waitForNextPeriodInterruptible()`₅₄.

`java.lang.SecurityException` - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setSchedulingParameters(
    javax.realtime.SchedulingParameters112 scheduling)
```

Description copied from interface: `javax.realtime.Schedulable`₈₁

Sets the scheduling parameters associated with this instance of `Schedulable`.

Since this affects the scheduling parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Specified By: `setSchedulingParameters`₉₅ in interface `Schedulable`₈₁

Parameters:

`scheduling` - A reference to the `SchedulingParameters`₁₁₂ object. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

Throws:

`java.lang.IllegalArgumentException` - Thrown when scheduling is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and scheduling is located in heap memory.

`IllegalAssignmentError448` - Thrown if this object cannot hold a reference to scheduling or scheduling cannot hold a reference to this.

```
public boolean setSchedulingParametersIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling)
```

Description copied from interface: `javax.realtime.Schedulable81`

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setSchedulingParametersIfFeasible96` in interface `Schedulable81`

Parameters:

scheduling - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if `this` cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to `this`.

```
public static void sleep(
    javax.realtime.Clock352 clock,
    javax.realtime.HighResolutionTime314 time)
    throws InterruptedException
```

A sleep method that is controlled by a generalized clock. Since the time is expressed as a `HighResolutionTime`₃₁₄, this method is an accurate timer with nanosecond granularity. The actual resolution available for the clock and even the quantity it measures depends on `clock`. The time base is the given `Clock`₃₅₂. The sleep time may be relative or absolute. If relative, then the calling thread is blocked for the amount of time given by `time`, and measured by `clock`. If absolute, then the calling thread is blocked until the indicated value is reached by `clock`. If the given absolute time is less than or equal to the current value of `clock`, the call to sleep returns immediately.

It is permissible to call `sleep` when control is in an `AsyncEventHandler`₃₉₃. The method cause the handler to sleep.

This method must not throw `IllegalAssignmentError`. It must tolerate `time` instances that may not be stored in `this`.

Parameters:

`clock` - The instance of `Clock`₃₅₂ used as the base. If `clock` is null the real-time clock (see `Clock.getRealtimeClock()`₃₅₃) is used. If `time` uses a time-base other than `clock`, `time` is reassociated with `clock` for purposes of this method.

`time` - The amount of time to sleep or the point in time at which to awaken.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()`₃₆ or `AsynchronouslyInterruptedException.fire()`₄₂₅ during the time between calling this method and returning from it.

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

`java.lang.UnsupportedOperationException` - Thrown if the sleep operation is not supported by clock.

`java.lang.IllegalArgumentException` - Thrown if `time` is null, or if `time` is a relative time less than zero.

```
public static void sleep(
    javax.realtime.HighResolutionTime314 time)
    throws InterruptedException
```

A sleep method that is controlled by a generalized clock. Since the time is expressed as a `HighResolutionTime314`, this method is an accurate timer with nanosecond granularity. The actual resolution available for the timer and even the quantity it measures depends on the clock associated with `time`. The sleep time may be relative or absolute. If relative, then the calling thread is blocked for the amount of time given by `time`, and measured by the clock associated with `time`. If absolute, then the calling thread is blocked until the indicated value is reached by the associated clock. If the given absolute time is less than or equal to the current value of the clock, the call to sleep returns immediately.

It is permissible to call `sleep` when control is in an `AsyncEventHandler393`. The method cause the handler to sleep.

This method must not throw `IllegalAssignmentError`. It must tolerate `time` instances that may not be stored in this.

Parameters:

`time` - The amount of time to sleep or the point in time at which to awaken.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()36` or `AsynchronouslyInterruptedException.fire()425` during the time between calling this method and returning from it.

`java.lang.ClassCastException` - Thrown if the current execution context is that of a Java thread.

`java.lang.UnsupportedOperationException` - Thrown if the sleep operation is not supported using the clock associated with `time`.

`java.lang.IllegalArgumentException` - Thrown if `time` is null, or if `time` is a relative time less than zero.

public void start()

Set up the real-time thread's environment and start it. The set up might include delaying it until the assigned start time and initializing the thread's scope stack. (See [ScopedMemory₁₇₂](#).)

Overrides: start in class Thread

public static boolean waitForNextPeriod()

Causes the current real-time thread to delay until the beginning of the next period. Used by threads that have a reference to a [ReleaseParameters₁₁₆](#) type of [PeriodicParameters₁₂₂](#) to block until the start of each period. The first period starts when this thread is first released. Each time it is called this method will block until the start of the next period unless the thread is in a deadline miss condition. In that case the operation of `waitForNextPeriod` is controlled by this thread's scheduler. (See *Priority Scheduler*.)

Returns: True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if this does not have a reference to a [ReleaseParameters₁₁₆](#) type of [PeriodicParameters₁₂₂](#).

`java.lang.ClassCastException` - Thrown if the current thread is not an instance of `RealtimeThread`.

Since: 1.0.1 Changed from an instance method to a static method.

**public static boolean waitForNextPeriodInterruptible()
throws InterruptedException**

The `waitForNextPeriodInterruptible()` method is a duplicate of [waitForNextPeriod\(\)₅₄](#) except that `waitForNextPeriodInterruptible` is able to throw `InterruptedException`.

Used by threads that have a reference to a [ReleaseParameters₁₁₆](#) type of [PeriodicParameters₁₂₂](#) to block until the start of each period. The first period starts when this thread is first released. Each time it is called this method will block until the start of the next period unless the thread is in a deadline miss condition. In that case the operation of `waitForNextPeriodInterruptible` is controlled by this thread's scheduler. (See *Priority Scheduler*.)

Returns: True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()`₃₆ or `AsynchronouslyInterruptedException.fire()`₄₂₅ during the time between calling this method and returning from it.

An interrupt during `waitForNextPeriodInterruptible` is treated as a release for purposes of scheduling. This is likely to disrupt proper operation of the periodic thread. The periodic behavior of the thread is unspecified until the state is reset by altering the thread's periodic parameters.

`java.lang.ClassCastException` - Thrown if the current thread is not an instance of `RealtimeThread`.

`java.lang.IllegalThreadStateException` - Thrown if this does not have a reference to a `ReleaseParameters`₁₁₆ type of `PeriodicParameters`₁₂₂.

Since: 1.0.1

5.2 NoHeapRealtimeThread

Declaration

`public class NoHeapRealtimeThread` extends `RealtimeThread`₂₉

All Implemented Interfaces: `java.lang.Runnable`, `Schedulable`₈₁

Description

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`₂₉. Because an instance of `NoHeapRealtimeThread` may immediately preempt any implemented garbage collector, logic contained in its `run()` is never allowed to allocate or reference any object allocated in the heap. At the byte-code level, it is illegal for a reference to an object allocated in heap to appear on a no-heap real-time thread's operand stack.

Thus, it is always safe for a `NoHeapRealtimeThread` to interrupt the garbage collector at any time, without waiting for the end of the garbage collection cycle or a defined preemption point. Due to these restrictions, a `NoHeapRealtimeThread` object

must be placed in a memory area such that thread logic may unexceptionally access instance variables and such that Java methods on `java.lang.Thread` (e.g., `enumerate` and `join`) complete normally except where execution would cause access violations. The constructors of `NoHeapRealtimeThread` require a reference to [ScopedMemory₁₇₂](#) or [ImmortalMemory₁₆₈](#).

When the thread is started, all execution occurs in the scope of the given memory area. Thus, all memory allocation performed with the `new` operator is taken from this given area.

5.2.1 Constructors

```
public NoHeapRealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.MemoryArea161 area)
```

Create a `NoHeapRealtimeThread`. This constructor is equivalent to `NoHeapRealtimeThread(scheduling, null, null, area, null, null)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the parameters are not compatible with the associated scheduler, if `area` is null, if `area` is heap memory, if `area` or `scheduling` is allocated in heap memory, or if this is in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if the new `NoHeapRealtimeThread` instance cannot hold a reference to `scheduling` or `area`, or if `scheduling` cannot hold a reference to the new `NoHeapRealtimeThread`.

```
public NoHeapRealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryArea161 area)
```

Create a no-heap real-time thread with the given characteristics. This constructor is equivalent to `NoHeapRealtimeThread(scheduling, release, null, area, null, null)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the parameters are not compatible with the associated scheduler, if `area` is null, if `area` is heap memory, if `area`, `release` or

scheduling is allocated in heap memory, or if this is in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if the new `NoHeapRealtimeThread` instance cannot hold a reference to non-null values of `scheduling`, `release` and `area`, or if `scheduling` and `release` cannot hold a reference to the new `NoHeapRealtimeThread`.

```
public NoHeapRealtimeThread(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.MemoryArea161 area,
    javax.realtime.ProcessingGroupParameters143 group,
    java.lang.Runnable logic)
```

Create a real-time thread with the given characteristics and a `java.lang.Runnable`. The thread group of the new thread is (effectively) null.

The newly-created no-heap real-time thread is associated with the scheduler in effect during execution of the constructor.

Parameters:

`scheduling` - The [SchedulingParameters₁₁₂](#) associated with this (and possibly other instances of [Schedulable₈₁](#)). 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 [ReleaseParameters₁₁₆](#) associated with this (and possibly other instances of [Schedulable₈₁](#)). If `release` is null the it defaults to the a copy of the creator's release parameters created in the same memory area as the new `NoHeapRealtimeThread`.

`memory` - The [MemoryParameters₂₇₃](#) associated with this (and possibly other instances of [Schedulable₈₁](#)). If `memory` is null, the new `NoHeapRealtimeThread` will have a null value for its memory parameters, and the amount or rate of memory allocation is unrestricted.

`area` - The [MemoryArea₁₆₁](#) associated with this. If `area` is null, an `java.lang.IllegalArgumentException` is thrown.

`group` - The [ProcessingGroupParameters₁₄₃](#) associated with this (and possibly other instances of [Schedulable₈₁](#)). If null, the new `NoHeapRealtimeThread` will not be associated with any processing group.

`logic` - The `Runnable` object whose `run()` method will serve as the logic for the new `NoHeapRealtimeThread`. If `logic` is null, the `run()` method in the new object will serve as its logic.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the parameters are not compatible with the associated scheduler, if `area` is null, if `area` is heap memory, if `area`, `scheduling release`, `memory` or `group` is allocated in heap memory, if `this` is in heap memory, or if `logic` is in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if the new `NoHeapRealtimeThread` instance cannot hold references to non-null values of the `scheduling release`, `memory` and `group`, or if those parameters cannot hold a reference to the new `NoHeapRealtimeThread`. Also thrown if `area` or `logic` cannot be stored in the new `RealtimeThread` object

5.2.2 Methods

`public void start()`

Checks if the `NoHeapRealtimeThread` is startable and starts it if it is.

Overrides: [start₅₄](#) in class [RealtimeThread₂₉](#)

Chapter 6

Scheduling

This section describes classes that control scheduling. These classes:

- Allow the definition of schedulable objects.
- Manage the assignment of execution eligibility to schedulable objects.
- Perform feasibility analysis for a set of schedulable objects.
- Control the admission of new schedulable objects to the feasibility set.
- Manage the execution of instances of the `AsyncEventHandler` and `Realtime-Thread` classes.
- Assign release characteristics to schedulable objects.
- Assign execution eligibility values to schedulable objects.
- Manage the execution of groups of schedulable objects that collectively exhibit additional release characteristics.

Definitions and Abbreviations

Schedulable objects include three execution states: *executing*, *blocked*, and *eligible-for-execution*.

- *Executing* refers to the state where the SO is currently running on a processor.
- *Blocked* refers to the state where the SO is not among those SO's which could be selected to have their state changed to executing. The blocked state will have a

reason associated with it, e.g., blocked-for-I/O-completion, blocked-for-release-event, or blocked-by-cost-overflow.

- *Eligible-for-execution* refers to the state where the SO could be selected to have its state changed to executing.

Each type of schedulable object defines its own *release events*, for example, the release events for a periodic SO occur with the passage of time.

Release is the changing of the state of a schedulable object from blocked-for-release-event to eligible-for-execution. If the state of an SO is blocked-for-release-event when a release event occurs then the state of the SO is changed to eligible-for-execution. Otherwise, a state transition from blocked-for-release-event to eligible-for-execution is queued—this is known as a *pending release*. When the next transition of the SO into state blocked-for-release-event occurs, and there is a pending release, the state of the SO is immediately changed to eligible-for-execution. (Some actions implicitly clear any pending releases.)

Completion is the changing of the state of a schedulable object from executing to blocked-for-release-event. Each completion corresponds to a release. A real-time thread is deemed to complete its most recent release when it terminates.

Deadline refers to a time before which a schedulable object expects to complete. The i^{th} deadline is associated with the i^{th} release event and a *deadline miss* occurs if the i^{th} completion would occur after the i^{th} deadline.

Deadline monitoring is the process by which the implementation responds to deadline misses. If a deadline miss occurs for a schedulable object, the deadline miss handler, if any, for that SO is released. This behaves as if there were an asynchronous event associated with the SO, to which the miss handler was bound, and which was fired when the deadline miss occurred.

Periodic, *sporadic*, and *aperiodic* are adjectives applied to schedulable objects which describe the temporal relationship between consecutive release events. Let R_i denote the time at which an SO has had the i^{th} release event occur. Ignoring the effect of release jitter:

- An SO is periodic when there exists a value $T > 0$ such that for all i , $R_{i+1} - R_i = T$, where T is called the period.
- An SO that is not periodic is said to be aperiodic.
- An aperiodic SO is said to be sporadic when there is a known value $T > 0$ such that for all i , $R_{i+1} - R_i \geq T$. T is then called the minimum interarrival time (MIT).

The *cost* of a schedulable object is an estimate of the maximum amount of CPU time that the SO requires between a release and its associated completion.

The *current CPU consumption* of a schedulable object is the amount of CPU time that the SO has consumed since its last release.

A *cost overrun* occurs when the schedulable object's current CPU consumption becomes greater than, or equal to, its cost.

Cost monitoring is the process by which the implementation tracks CPU consumption and responds to cost overruns. If a cost overrun occurs for a schedulable object, the cost overrun handler, if any, for that SO is released. This behaves as if there were an asynchronous event associated with the SO, to which the overrun handler was bound, and which was fired when the cost overrun occurred. (Cost monitoring is an optional facility in an implementation of the RTSJ.)

The *base priority* of a schedulable object is the priority given in its associated `PriorityParameters` object; the base priority of a Java thread is the priority returned by its `getPriority` method.

When it is not in the enforced state, the *active priority* of a schedulable object or a Java thread is the maximum of its base priority and any priority it has acquired due to the action of priority inversion avoidance algorithms (see the *Synchronization Chapter*),

A *processing group* is a collection of schedulable objects whose combined execution has further time constraints which the scheduler uses to govern the group's execution eligibility.

A *scheduler* manages the execution of schedulable objects: it detects deadline misses, and performs admission control and cost monitoring. It also manages the execution of Java threads.

The *base scheduler* is an instance of the `PriorityScheduler` class as defined in this specification. This is the initial default scheduler.

Overview

The scheduler required by this specification is fixed-priority preemptive with at least 28 unique priority levels. It is represented by the class `PriorityScheduler` and is called the *base scheduler*.

The schedulable objects required by this specification are defined by the classes `RealtimeThread`, `NoHeapRealtimeThread`, `AsyncEventHandler` and `BoundAsyncEventHandler`. The base scheduler assigns processor resources according to the schedulable objects' release characteristics, execution eligibility, and processing group values. Subclasses of the schedulable objects are also schedulable objects and behave as these required classes.

An instance of the `SchedulingParameters` class contains values of execution eligibility. A schedulable object is considered to have the execution eligibility represented by the `SchedulingParameters` object currently bound to it. For

implementations providing only the base scheduler, the scheduling parameters object is an instance of `PriorityParameters` (a subclass of `SchedulingParameters`).

An instance of the `ReleaseParameters` class or its subclasses, `PeriodicParameters`, `AperiodicParameters`, and `SporadicParameters`, contains values that define a particular release characteristic. A schedulable object is considered to have the release characteristics of a single associated instance of the `ReleaseParameters` class. In all cases the base scheduler uses these values to perform its feasibility analysis over the set of schedulable objects and admission control for the schedulable object.

For a real-time thread the scheduler defines the behavior of the real-time thread's `waitForNextPeriod` and `waitForNextPeriodInterruptible` methods, and monitors cost overrun and deadline miss conditions based on its release parameters. For asynchronous event handlers, the scheduler monitors cost overruns and deadline misses.

Release parameters also govern the treatment of the minimum interarrival time for sporadic schedulable objects.

An instance of the `ProcessingGroupParameters` class contains values that define a temporal scope for a processing group. If a schedulable object has an associated instance of the `ProcessingGroupParameters` class, it is said to execute within the temporal scope defined by that instance. A single instance of the `ProcessingGroupParameters` class can be (and typically is) associated with many SO's. If the implementation supports cost monitoring, the combined processor demand of all of the SO's associated with an instance of the `ProcessingGroupParameters` class must not exceed the values in that instance (i.e., the defined temporal scope). The processor demand is determined by the Scheduler.

Semantics and Requirements

This section establishes the semantics and requirements that are applicable across the classes of this chapter, and also defines the required scheduling algorithm. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

Semantics and Requirements Governing all Schedulers

1. Schedulers other than the base scheduler may change the execution eligibility of the schedulable objects which they manage according to their scheduling algorithm.
2. If an implementation provides any public schedulers other than the base scheduler it shall provide documentation describing each scheduler's semantics in language

and constructs appropriate to the provided scheduling algorithms. This documentation must include the list of classes that constitute schedulable objects for the scheduler unless that list is the same as the list of schedulable objects for the base scheduler.

3. This specification does not require any particular feasibility algorithm be implemented in the `Scheduler` object. The default algorithm always returns success for sporadic and periodic schedulable objects, as it assumes adequate resources, but it always returns false for aperiodic schedulable objects since no pool of resources would render such a load feasible.
4. Implementations that provide a scheduler with a feasibility algorithm other than the default are required to document the behavior of that algorithm and any assumptions it makes.

Semantics and Requirements Governing the Base Scheduler

The semantics for the base scheduler assume a uni-processor execution environment. While implementations of the RTSJ are not precluded from supporting multi-processor execution environments, no explicit consideration for such environments has been given in this specification.

The base scheduler supports the execution of all schedulable objects and Java threads, but it only controls the release of periodic real-time threads, and aperiodic asynchronous event handlers.

Priorities

The execution scheduling semantics described in this section are defined in terms of a conceptual model that contains a set of queues of schedulable objects that are eligible for execution. There is, conceptually, one queue for each priority. No implementation structures are necessarily implied by the use of this conceptual model. It is assumed that no time elapses during operations described using this model, and therefore no simultaneous operations are possible.

1. The base scheduler must support at least 28 distinct values (real-time priorities) that can be stored in an instance of `PriorityParameters` in addition to the values 1 through 10 required to support the priorities defined by `java.lang.Thread`. The real-time priority values must be greater than 10, and they must include all integers from the base scheduler's `getMinPriority()` value to its `getMaxPriority()` value inclusive. The 10 priorities defined for `java.lang.Thread` must effectively have lower execution eligibility than the real-time priorities, but beyond this, their behavior is as defined by the specification of `java.lang.Thread`.
2. Higher priority values in an instance of `PriorityParameters` have a higher execution eligibility.

3. Assignment of any of the real-time priority values to any schedulable object controlled by the base priority scheduler is legal. It is the responsibility of application logic to make rational priority assignments.
4. If two schedulable objects have different active priorities, the schedulable object with the higher active priority will always execute in preference to the schedulable object with the lower value when both are eligible for execution.
5. A schedulable object that is executing will continue to execute until it either blocks, or is preempted by a higher-priority schedulable object.
6. The base scheduler does not use the importance value in the ImportanceParameters subclass of PriorityParameters.
7. The dispatching mechanism must allow the preemption of the execution of schedulable objects and Java threads at a point not governed by the preempted object.
8. For schedulable objects managed by the base scheduler the implementation must not change the execution eligibility for any reason other than
 - a. Implementation of a priority inversion avoidance algorithm or
 - b. As a result of a program's request to change the priority parameters associated with one or more schedulable objects; e.g., by changing a value in a scheduling parameter object that is used by one or more schedulable objects, or by using `setSchedulingParameters()` to give a schedulable object a different `SchedulingParameters` value.
9. Use of `Thread.setPriority()`, any of the methods defined for schedulable objects, or any of the methods defined for parameter objects must not affect the correctness of the priority inversion avoidance algorithms controlled by `PriorityCeilingEmulation` and `PriorityInheritance` - see the *Synchronization* chapter.
10. A schedulable object that is preempted by a higher priority schedulable object is placed in the queue for its active priority, at a position determined by the implementation. The implementation must document the algorithm used for such placement. It is recommended that a preempted schedulable object be placed at the front of the appropriate queue.
11. A real-time thread that performs a `yield()` is placed at the tail of the queue for its active priority level.
12. A blocked schedulable object that becomes eligible for execution is added to the tail of the queue for that priority. This behavior also applies to the initial release of a schedulable object.
13. For a schedulable object whose active priority is changed as a result of explicitly setting its base priority (through `PriorityParameters.setPriority()` method, `RealtimeThread.setSchedulingParameters()` method, or `Thread.set-`

Priority() method), this schedulable object is added to the tail of the queue for its new priority level. Queuing when priorities are adjusted by priority inversion avoidance algorithms is governed by semantics specified in the *Synchronization* chapter.

14. If schedulable object *A* managed by the base scheduler creates a Java thread, *B*, then the initial base priority of *B* is the priority value returned by the `getMaxPriority` method of *B*'s `java.lang.ThreadGroup` object.
15. For real-time threads managed by the base scheduler, priority limits set by `java.lang.ThreadGroup` objects are not enforced.
16. `PriorityScheduler.getNormPriority()` shall be set to:

$$((\text{PriorityScheduler.getMaxPriority()} - \text{PriorityScheduler.getMinPriority()})/3) + \text{PriorityScheduler.getMinPriority}()$$

Parameter Values

The scheduler uses the values contained in the different parameter objects associated with a schedulable object to control the behavior of the schedulable object. The scheduler determines what values are valid for the schedulable objects it manages, which defaults apply and how changes to parameter values are acted upon by the scheduler. Invalid parameter values result in exceptions, as documented in the relevant classes and methods.

1. The default values for the base scheduler are:
 - a. Scheduling parameters are copied from the creating SO if possible; if the creating SO does not have scheduling parameters the default is an instance of the default priority parameters value.
 - b. Release parameters default to an instance of the default aperiodic parameters (see `AperiodicParameters`).
 - c. Memory parameters default to null which signifies that memory allocation by the schedulable object is not constrained by the scheduler.
 - d. Processing group parameters default to null which signifies that the schedulable object is not a member of any processing group and is not subject to processing group based limits on processor utilization.
 - e. The default scheduling parameter values for parameter objects created by an SO controlled by the base scheduler are: (see `PriorityScheduler`)

Attribute	Default Value
<i>Priority parameters</i>	
priority	norm priority
<i>Importance parameters</i>	
importance	No default. A value must be supplied.

2. All numeric or `RelativeTime` attributes in parameter values must be greater than or equal to zero.
3. Values of period must be greater than zero.
4. Deadline values in `ReleaseParameters` objects must be less than or equal to their period values (where applicable), but the deadline may be greater than the minimum interarrival time in a `SporadicParameters` object.
5. Changes to scheduling, release, memory, and processing group parameters (by methods on the schedulable objects bound to the parameters or by altering the parameter objects themselves) have two effects:
 - a. They immediately affect the feasibility test of the scheduler.
 - b. They potentially modify the behavior of the scheduler with regard to those schedulable objects. When such changes in behavior take effect depends on the parameter in question, and the type of schedulable object, as described below.
6. Changes to scheduling, release, memory, and processing group parameters are acted upon by the base scheduler as follows:
 - a. Changes to scheduling parameters take effect immediately except as provided by priority inversion avoidance algorithms.
 - b. Changes to release parameters depend on the parameter being changed, the type of release parameter object and the type of schedulable object:
 - i. Changes to the deadline and the deadline miss handler take effect at each release event as follows: if the i th release event occurred at a time t_i , then the i th deadline is the time $t_i + D_i$, where D_i is the value of the deadline stored in the schedulable object's release parameters object at the time t_i . If a deadline miss occurs then it is the deadline miss handler that was installed in the schedulable object's release parameters at time t_i that is released.
 - ii. Changes to cost and the cost overrun handler take effect immediately.

- iii. Changes to the period and start time values in `PeriodicParameters` objects are described in “Periodic Release of Real-time Threads” below. (The base scheduler does not manage the release of periodic schedulable objects other than periodic real-time threads.)
- iv. Changes to the additional values in `AperiodicParameters` objects and `SporadicParameters` are described, respectively, in “Aperiodic Release Control” and “Sporadic Release Control”, below. (The base scheduler does not manage the release of aperiodic schedulable objects other than aperiodic asynchronous event handlers.)
- v. Changes to the type of release parameters object generally take effect after completion, except as documented in the following sections.
- c. Changes to memory parameters take effect immediately.
- d. Changes to processing group parameters take effect as described in “Processing Groups” below.
- e. Changes to the scheduler responsible for a schedulable object take effect at completion.

Cost Monitoring

Cost monitoring is an optional facility in the implementation of the RTSJ, but when supported it must conform to the requirements and definitions as presented in this section.

1. The cost of an SO is defined by the value returned by invoking the `getCost` method of the SO’s release parameters object.
2. When an SO is initially released it’s current CPU consumption is zero and as the SO executes, the current CPU consumption increases. The current CPU consumption is set to zero in response to certain actions as described below.
3. If at any time, due to either execution of the SO or a change in the SO’s cost, the current CPU consumption becomes greater than, or equal to, the current cost of the SO, then a cost overrun is triggered. The implementation is required to document the granularity at which the current CPU consumption is updated.
4. When a cost overrun is triggered, the cost overrun handler associated with the SO, if any, is released. If the most recent release of the SO is the *ith* release, and the *i+1* release event has not yet occurred, then:
 - a. If the state of the SO is either executing or eligible-for-execution, then the SO is placed into the state blocked-by-cost-overrun. There may be a bounded delay between the time at which a cost overrun occurs and the time at which the SO becomes blocked-by-cost-overrun.
 - b. Otherwise, the SO must have been blocked for a reason other than blocked-

by-cost-overflow. In this case, the state change to blocked-by-cost-overflow is left pending: if the blocking condition for the SO is removed, then its state changes to blocked-by-cost-overflow. There may be a bounded delay between the time at which the blocking condition is removed and the time at which the SO becomes blocked-by-cost-overflow.

Otherwise, if the $i+1$ release event has occurred, the current CPU consumption is set to zero, the SO remains in its current state and the cost monitoring system considers the most recent release to now be the $i+1$ release.

5. When the i th release event occurs for an SO, the action taken depends on the state of the SO:
 - a. If the SO is blocked-by-cost-overflow then the cost monitoring system considers the most recent release to be the i th release, the current CPU consumption is set to zero and the SO is made eligible for execution;
 - b. Otherwise, if the SO is blocked for a reason other than blocked-by-cost-overflow then:
 - i. If there is a pending state change to blocked-by-cost-overflow then: the pending state change is removed, the cost monitoring system considers the most recent release to be the i th release, the current CPU consumption is set to zero and the SO remains in its current blocked state;
 - ii. Otherwise, no cost monitoring action occurs.
 - c. Otherwise no cost monitoring action occurs.
6. When the i th release of an SO completes, and the cost monitoring system considers the most recent release to be the i th release, then the current CPU consumption is set to zero and the cost monitoring system considers the most recent release to be the $i+1$ release. Otherwise, no cost monitoring action occurs.
7. Changes to the cost parameter take effect immediately:
 - a. If the new cost is less than or equal to the current CPU consumption, and the old cost was greater than the current CPU consumption, then a cost overflow is triggered.
 - b. If the new cost is greater than the current CPU consumption:
 - i. If the SO is blocked-by-cost-overflow, then the SO is made eligible for execution;
 - ii. Otherwise, if the SO is blocked for a reason other than blocked-by-cost-overflow, and there is a pending state change to blocked-by-cost-overflow, then the pending state change is removed;
 - iii. Otherwise, no cost monitoring action occurs.
8. The state of the cost monitoring system for an SO can be *reset* by the scheduler

(see 5c in the Periodic Release of Real-time Threads section, below). If the most recent release of the SO is considered to be the m th release, and the most recent release event for the SO was the n th release event (where $n > m$), then a reset causes the cost monitoring system to consider the most recent release to be the n th release, and to zero the current CPU consumption.

Periodic Release of Real-time Threads

A schedulable object with release parameters of type `PeriodicParameters` is expected to be released periodically. For asynchronous event handlers this would occur if the associated asynchronous event fired periodically. For real-time threads periodic release behavior is achieved by executing in a loop and invoking the `RealtimeThread.waitForNextPeriod` method, or its interruptible equivalent `RealtimeThread.waitForNextPeriodInterruptible` within that loop. For simplicity, unless otherwise stated, the semantics in this section apply to both forms of that method.

1. A periodic real-time thread's release characteristics are determined by the following:
 - a. The invocation of the real-time thread's start method.
 - b. The action of the `RealtimeThread` methods `waitForNextPeriod`, `waitForNextPeriodInterruptible`, `schedulePeriodic` and `deschedulePeriodic`;
 - c. The occurrence of deadline misses and whether or not a miss handler is installed; and
 - d. The passing of time that generates periodic release events
2. The *initial release event* of a periodic real-time thread occurs in response to the invocation of the its start method, in accordance with the start time specified in its release parameters - see `PeriodicParameters`.
3. Changes to the start time in a real-time thread's `PeriodicParameters` object only have an effect on its initial release time. Consequently, if a `PeriodicParameters` object is bound to multiple real-time threads, a change in the start time may affect all, some or none, of those threads, depending on whether or not start has been invoked on them.
4. Subsequent release events occur as each period falls due, except as described below in 5(e), at times determined as follows: if the i th release event occurred at a time t_i , then the $i+1$ release event occurs at the time t_i+T_i , where T_i is the value of the period stored in the real-time thread's `PeriodicParameters` object at the time t_i .
5. The implementation should behave effectively as if the following state variables were added to a real-time thread's state, and manipulated by the actions in (1) as

described below:

boolean `descheduled`, integer `pendingReleases`, integer `missCount`, and boolean `lastReturn`.

- a. Initially: `descheduled = false`, `pendingReleases = 0`, `missCount = 0`, and `lastReturn = true`.
- b. When the real-time thread's `deschedulePeriodic` method is invoked: set the value of `descheduled` to true.
- c. When the real-time thread's `schedulePeriodic` method is invoked: set the value of `descheduled` to false; then if the thread is blocked-for-release-event, set the value of `pendingReleases` to zero, and tell the cost monitoring system to reset for this thread.
- d. When `descheduled` is true, the real-time thread is said to be *descheduled*.
- e. A real-time thread that has been descheduled and is blocked-for-release-event will not receive any further release events until after it has been rescheduled by a call to `schedulePeriodic`; this means that no deadline misses can occur until the thread has been rescheduled. The descheduling of a real-time thread has no effect on its initial release.
- f. When each period is due:
 - i. If the state of the real-time thread is blocked-for-release-event (that is, it is waiting in `waitForNextPeriod`), then if the thread is descheduled then do nothing, else increment the value of `pendingReleases`, inform cost monitoring that the next release event has occurred, and notify the thread to make it eligible for execution;
 - ii. Otherwise, increment the value of `pendingReleases`, and inform cost monitoring that the next release event has occurred.
- g. On each deadline miss:
 - i. If the real-time thread has a deadline miss handler: set the value of `descheduled` to true, atomically release the handler with its `fireCount` increased by the value of `missCount+1` and zero `missCount`;
 - ii. Otherwise add one to the `missCount` value.
- h. When the `waitForNextPeriod` method is invoked by the current real-time thread there are two possible behaviors depending on the value of `missCount`:
 - i. If `missCount` is greater than zero: decrement the `missCount` value; then if the `lastReturn` value is false, completion occurs: apply any pending parameter changes, decrement `pendingReleases`, inform cost monitoring the real-time thread has completed and return false; otherwise set the

lastReturn value to false and return false.

- ii. Otherwise, apply any pending parameter changes, inform cost monitoring of completion, and then wait while `descheduled` is true, or `pendingReleases` is zero. Then set the `lastReturn` value to true, decrement `pendingReleases`, and return true.
6. An invocation of the `waitForNextPeriodInterruptible` method behaves as described above with the following additions:
- a. If the invocation commences when an instance of `AsynchronouslyInterruptedException` (AIE) is pending on the real-time thread, then the invocation immediately completes abruptly by throwing that pending instance as an `InterruptedException`. If this occurs, the most recent release has not completed. If the pending instance is the generic AIE instance then the interrupt state of the real-time thread is cleared.
 - b. If an instance of AIE becomes pending on the real-time thread while it is blocked-for-release-event, and the real-time thread is descheduled, then the AIE remains pending until the real-time thread is no longer descheduled. Execution then continues as in (c).
 - c. If an instance of AIE becomes pending on the real-time thread while it is blocked-for-release-event, and it is not descheduled, then this acts as a release event:
 - i. The real-time thread is made eligible for execution.
 - ii. Upon execution the invocation completes abruptly by throwing the pending AIE instance as an `InterruptedException`. If the pending instance is the generic AIE instance then the interrupt state of the real-time thread is cleared.
 - iii. If the AIE becomes pending at a time t_{int} then:
 - The deadline associated with this release is the time $t_{int} + D_{int}$, where D_{int} is the value of the deadline stored in the real-time thread's release parameters object at the time t_{int} .
 - The next release time for the real-time thread will be $t_{int} + T_{int}$, where T_{int} is the value of the period stored in the real-time thread's release parameters object at the time t_{int} .
 - iv. Cost monitoring is informed of the release event

When the thrown AIE instance is caught, the AIE becomes pending again (as per the usual semantics for AIE) until it is explicitly cleared.

7. If an aperiodic real-time thread has its release parameters set to periodic parame-

ters, then calls `waitForNextPeriod`, the change from non-periodic to periodic scheduling effectively takes place between the call to `waitForNextPeriod` and the first periodic release. The first periodic release is determined by the start time specified in the real-time thread's periodic parameters. If that start time is an absolute time in the future, then that is the first periodic release time; if it is an absolute time in the past then the time at which `waitForNextPeriod` was called is the first periodic release time and the release occurs immediately. If the start time is a relative time, then it is relative to the time at which `waitForNextPeriod` was called; if that time is in the past then the release occurs immediately.

8. If a periodic real-time thread has its release parameters set to be other than an instance of `PeriodicParameters` then the change from periodic to non-periodic scheduling effectively takes place immediately, unless the thread is blocked-for-release-event, in which case the change takes place after the next release event. When this change occurs, the deadline for the real-time thread is that which was in effect for the most recent release.

Pseudo-Code for Periodic Thread Actions

The semantics of the previous section can be more clearly understood by viewing them in pseudo-code form for each of the methods and actions involved. In the following no mechanism for blocking and unblocking a thread is prescribed. The use of the wait and notify terminology in places is purely an aid to expressing the desired semantics in familiar terms.

```
// These values are part of thread state.
boolean descheduled = false;
int pendingReleases = 0;
boolean lastReturn = true;
int missCount = 0;
deschedulePeriodic(){
    descheduled = true;
}
schedulePeriodic(){
    descheduled = false;
    if (blocked-for-release-event) {
        pendingReleases = 0;
        costMonitoringReset();
    }
}
onNextPeriodDue(){
    if (blocked-for-release-event) {
        if (descheduled) {
            ; // do nothing
        }
        else {
            pendingReleases++;
            notifyCostMonitoringOfReleaseEvent();
            notify it; // make eligible for execution
        }
    }
    else {
        pendingReleases++;
        notifyCostMonitoringOfReleaseEvent();
    }
}
}
```

```

onDeadlineMiss(){
    if (there is a miss handler) {
        descheduled = true;
        release miss handler with fireCount increased by missCount+1
        missCount = 0;
    }
    else {
        missCount++;
    }
}
}
waitForNextPeriod{
    assert(pendingReleases >= 0);
    if (missCount > 0 ) {
        // Missed a deadline without a miss handler
        missCount--;
        if (lastReturn == false) {
            // Changes "on completion" take place here
            performParameterChanges();
            pendingReleases--;
            notifyCostMonitoringOfCompletion();
        }
        lastReturn = false;
        return false;
    }
    else {
        // Changes "on completion" take place here
        performParameterChanges();
        notifyCostMonitoringOfCompletion();
        wait while (descheduled || pendingReleases == 0); // blocked-
for-release-event
        pendingReleases--;
        lastReturn = true;
        return true;
    }
}
}

```

Aperiodic Release Control

Aperiodic schedulable objects are released in response to events occurring, such as the starting of a real-time thread, or the firing of an associated asynchronous event for an asynchronous event handler. The occurrence of these events, each of which is a potential release event, is termed an *arrival*, and the time that they occur is termed the *arrival time*.

The base scheduler behaves effectively as if it maintained a queue, called the arrival time queue, for each aperiodic schedulable object. This queue maintains information related to each release event from its "arrival" time until the associated release completes, or another release event occurs - whichever is later. If an arrival is accepted into the arrival time queue, then it is a release event and the time of the release event is the arrival time. The initial size of this queue is an attribute of the schedulable object's aperiodic parameters, and is set when the parameter object is associated with the SO. Over time the queue may become full and its behavior in this situation is determined by the queue overflow policy specified in the SO's aperiodic parameters. There are four overflow policies defined:

Policy	Action on Overflow
IGNORE	Silently ignore the arrival. The arrival is not accepted, no release event occurs, and, if the arrival was caused programmatically (such as by invoking <code>fire</code> on an asynchronous event), the caller is not informed that the arrival has been ignored.
EXCEPT	Throw an <code>ArrivalTimeQueueOverflowException</code> . The arrival is not accepted, and no release event occurs, but if the arrival was caused programmatically, the caller will have <code>ArrivalTimeQueueOverflowException</code> thrown.
REPLACE	The arrival is not accepted and no release event occurs. If the completion associated with the last release event in the queue has not yet occurred, and the deadline has not been missed, then the release event time for that release event is replaced with the arrival time of the new arrival. This will alter the deadline for that release event. If the completion associated with the last release event has occurred, or the deadline has already been missed, then the behavior of the REPLACE policy is equivalent to the IGNORE policy.
SAVE	Behave effectively as if the queue were expanded as necessary to accommodate the new arrival. The arrival is accepted and a release event occurs.

Under the SAVE policy the queue can grow and shrink over time.

Changes to the queue overflow policy take effect immediately. When an arrival occurs and the queue is full, the policy applied is the policy as defined at that time.

Aperiodic Real-time Threads

Aperiodic real-time threads executing under the base scheduler have the following characteristics:

1. The initial release event occurs when `start` is invoked upon it.
2. There are no subsequent release events.
3. Completion occurs only through termination.
4. When a deadline miss occurs, the deadline miss handler, if any, is released.

5. If a cost overrun occurs the overrun handler, if any, is released and the real-time thread is placed in the state blocked-by-cost-overrun. It can become eligible for execution again only through a change to its cost parameter.

Sporadic Release Control

Sporadic parameters include a minimum interarrival time, MIT, that characterizes the expected frequency of releases. When an arrival is accepted implementation behaves as if it calculates the earliest time at which the next arrival could be accepted, by adding the current MIT to the arrival time of this accepted arrival. The scheduler guarantees that each sporadic schedulable object it manages, is released at most once in any MIT. It implements two mechanisms for enforcing this rule:

- *Arrival-time regulation* controls the work-load by considering the time between arrivals. If a new arrival occurs earlier than the expected next arrival time then a MIT violation has occurred, and the scheduler acts to prevent a release from occurring that would break the “one release per MIT” guarantee. Three arrival-time MIT-violation policies are supported:

Policy	Action on Violation
IGNORE	Silently ignore the violating arrival. The arrival is not accepted, no release event occurs, and, if the arrival was caused programmatically (such as by invoking <code>fire</code> on an asynchronous event), the caller is not informed that the arrival has been ignored.
EXCEPT	Throw a <code>MITViolationException</code> . The arrival is not accepted, and no release event occurs, but if the arrival was caused programmatically, the caller will have <code>MITViolationException</code> thrown.
REPLACE	The arrival is not accepted and no release event occurs. If the completion associated with the last release event in the queue has not yet occurred, and the deadline has not been missed, then the release event time for that release event is replaced with the arrival time of the new arrival. This will alter the deadline for that release event. If the completion associated with the last release event has occurred, or the deadline has already been missed, then the behavior of the <code>REPLACE</code> policy is equivalent to the <code>IGNORE</code> policy.

- *Execution-time regulation* occurs if the MIT violation policy SAVE is in effect. Under this policy all arrivals are accepted, but the scheduler behaves effectively as if released schedulable objects were further constrained by a scheduling policy that restricts execution to at most one release per MIT. This policy is only able to delay the effective release of a schedulable object. The deadline of each release event is always set relative to its arrival time. This policy may not schedule the effective release of an async event handler until after its deadline has passed. In this case the deadline miss handler is released at the deadline time even though the related async event has not yet reached its effective release.

The SAVE policy makes no direct use of the next expected arrival time, but it maintains the value in case the MIT violation policy is changed from SAVE to one of the arrival-time regulation policies.

The *effective release time* of a release event i is the earliest time that the handler can be released in response to that release event. It is determined for each release event based on the MIT policy in force at the release event time:

- For IGNORE, EXCEPT and REPLACE the effective release time is the release event time.
- For SAVE the effective release time of release event i is the effective release time of release event $i-1$ plus the current value of the MIT.

The scheduler will delay the release associated with the release event at the head of the arrival time queue until the current time is greater than or equal to the effective release time of that release event.

Changes to minimum interarrival time and the MIT violation policy take effect immediately, but only affect the next expected arrival time, and effective release time, for release events that occur after the change.

Aperiodic and Sporadic Release Control for Asynchronous Event Handlers

Asynchronous event handlers can be associated with one or more asynchronous events. When an asynchronous event is fired, all handlers associated with it are released, according to the semantics below:

1. Each firing of an associated asynchronous event is an arrival. If the handler has release parameters of type `AperiodicParameters`, then the arrival may become a release event for the handler, according to the semantics given in “Aperiodic Release Control” above. If the handler has release parameters of type `SporadicParameters`, then the arrival may become a release event for the handler, according to the semantics given in “Sporadic Release Control” above. If the handler has release parameters of a type other than `SporadicParameters` then the arrival is a

- release event, and the arrival-time is the release event time.
2. For each release event that occurs for a handler, an entry is made in the arrival-time queue and the handler's `fireCount` is incremented by one.
 3. Initially a handler is considered to be blocked-for-release-event and its `fireCount` is zero.
 4. Releases of a handler are serialized by having its `handleAsyncEvent` method invoked repeatedly while its `fireCount` is greater than zero:
 - a. Before invoking `handleAsyncEvent`, the `fireCount` is decremented and the front entry (if still present) removed from the arrival-time queue.
 - b. Each invocation of `handleAsyncEvent`, in this way, is a release.
 - c. The return from `handleAsyncEvent` is the completion of a release.
 - d. Processing of any exceptions thrown by `handleAsyncEvent` occurs prior to completion.
 5. The deadline for a release is relative to the release event time and determined at the release event time according to the value of the deadline contained in the handler's release parameters. This value does not change, except as described previously for handlers using a REPLACE policy for MIT violation or arrival-time queue overflow.
 6. The application code can directly modify the `fireCount` as follows:
 - a. The `getAndDecrementPendingFireCount` method decreases the `fireCount` by one (if it was greater than zero), and returns the old value. This removes the front entry from the arrival-time queue but otherwise has no effect on the scheduling of the current schedulable object, nor the handler itself.
 - b. The `getAndClearPendingFireCount` method is functionally equivalent to invoking `getAndDecrementPendingFireCount` until it returns zero, and returning the original `fireCount` value.
 - c. The `getAndIncrementPendingFireCount` method attempts to increase the `fireCount` by one, and returns the old value. It behaves effectively as if a private event, associated only with this handler, were fired, in accordance with semantic (1) above. This pseudo-firing is treated as a normal firing with respect to the other semantics in this section.
 7. The scheduler may delay the invocation of `handleAsyncEvent` to ensure the effective release time honors any restrictions imposed by the MIT violation policy, if applicable, of that release event.
 8. Cost monitoring for an asynchronous event handler interacts with release events and completions as previously defined with the added requirement that at the completion of `handleAsyncEvent`, if the `fireCount` is now zero, then the cost

monitoring system is told to reset for this handler.

Processing Groups

A processing group is defined by a processing group parameters object, and each SO that is bound to that parameter object is called a *member* of that processing group.

Processing groups are only functional in a system that implements processing group enforcement. Although the processing group itself does not consume CPU time, it acts as a proxy for its members.

Definitions for Processing Groups

The *enforced priority* of a schedulable object is a priority with no execution eligibility.

Semantics for Processing Groups

1. The deadline of a processing group is defined by the value returned by invoking the `getDeadline` method of the processing group parameters object.
2. A deadline miss for the processing group is triggered if any member of the processing group consumes CPU time at a time greater than the deadline for the most recent release of the processing group.
3. When a processing group misses a deadline:
 - a. If the processing group has a miss handler, it is released for execution
 - b. If the processing group has no miss handler, no action is taken.
4. The cost of a processing group is defined by the value returned by invoking the `getCost` method of the processing group parameters object.
5. When a processing group is initially released, its current CPU consumption is zero and as the members of the processing group execute, the current CPU consumption increases. The current CPU consumption is set to zero in response to certain actions as described below.
6. If at any time, due to either execution of the members of the processing group or a change in the parameter group's cost, the current CPU consumption becomes greater than, or equal to, the current cost of the processing group, then a cost overrun is triggered. The implementation is required to document the granularity at which the current CPU consumption is updated.
7. When a cost overrun is triggered, the cost overrun handler associated with the processing group, if any, is released, and the processing group enters the *enforced state*. For each member of the processing group:
 - a. The SO is placed into the enforced state.
 - b. When a SO is in the enforced state the base scheduler schedules that SO effectively as if the enforced priority were used in place of the SO's base priority.

8. When the a release event occurs for a processing group, the action taken depends on the state of the processing group:
 - a. If the processing group is not in the enforced state then the current CPU consumption for the group is set to zero;
 - b. Otherwise the processing group is in the enforced state. It is removed from the enforced state, the current CPU consumption of the group is set to zero, and each member of the group is removed from the enforced state.
9. Changes to the cost parameter take effect immediately:
 - a. If the new cost is less than or equal to the current CPU consumption, and the old cost was greater than the current CPU consumption, then a cost overrun is triggered.
 - b. If the new cost is greater than the current CPU consumption:
 - i. If the processing group is enforced, then the processing group behaves as defined in semantic 8.
 - ii. Otherwise, no cost monitoring action occurs.
10. Changes to other parameters take place as follows:
 - a. Start: can only be changed before the parameters group is started; i.e., before the start time or before the parameter object is associated with any SO. Changes take effect immediately.
 - b. Period: at each release the next period is set based on the current value of the processing group's period.
 - c. Deadline: at each release the next deadline is set based on the current value of the processing group's deadline.
 - d. OverrunHandler: at each release the overrunHandler is set based on the current value of the processing group's overrunHandler.
 - e. MissHandler: at each release the missHandler is set based on the current value of the processing group's missHandler.
11. Changes to the membership of the processing group take effect immediately.
12. The start time for the processing group may be relative or absolute.
 - a. If the start time is absolute, the processing group behaves effectively as if the initial release time were the start time.
 - b. If the start time is relative, the initial release time is computed relative to the time `start` or `fire` (as appropriate) is first called for a member of the processing group.

Note: Until a processing group starts, its budget cannot be replenished, but its members will be enforced if they exceed the initial budget. Also, once a processing

group is started it behaves effectively as if it continued running continuously until the defining `ProcessingGroupParameters` object is freed.

Rationale

As specified the required semantics and requirements of this section establish a scheduling policy that is very similar to the scheduling policies found on the vast majority of real-time operating systems and kernels in commercial use today. The semantics and requirements for the base scheduler accommodate existing practice, which is a stated goal of the effort.

There is an important division between priority schedulers that force periodic context switching between tasks at the same priority, and those that do not cause these context switches. By not specifying *time slicing* behavior this specification calls for the latter type of priority scheduler. In POSIX terms, `SCHED_FIFO` meets the RTSJ requirements for the base scheduler, but `SCHED_RR` does not meet those requirements.

Although a system may not implement the first release (start) of a schedulable object as unblocking that schedulable object, under the base scheduler those semantics apply; i.e., the schedulable object is added to the tail of the queue for its active priority.

Some research shows that, given a set of reasonable common assumptions, 32 distinct priority levels are a reasonable choice for close-to-optimal scheduling efficiency when using the rate-monotonic priority assignment algorithm (256 priority levels provide better efficiency). This specification requires at least 28 distinct priority levels as a compromise noting that implementations of this specification will exist on systems with logic executing outside of the Java Virtual Machine and may need priorities above, below, or both for system activities.

In order not to undermine any feasibility analysis, the default behavior for implementations that support cost monitoring is that a schedulable object receives no more than `cost` units of CPU time during each release. The programmer must explicitly change the cost attribute to override the scheduler.

Cost enforcement may be deferred while the overrun schedulable object holds locks that are out of application control, such as locks used to protect garbage collection. Applications should include the resulting jitter in any analysis that depends on cost enforcement.

When a schedulable object is enforced because of cost overrun in a processing group the enforced priority is used for scheduling instead of the schedulable object's base priority. The enforced priority's application is limited. The enforced priority is not returned as the schedulable object's priority from methods such as `getPriority()`,

and the semantics of the active priority continue to operate when a schedulable object is enforced.

6.1 Schedulable

Declaration

```
public interface Schedulable extends java.lang.Runnable
```

All Superinterfaces: `java.lang.Runnable`

All Known Implementing Classes: [RealtimeThread₂₉](#), [AsyncEventHandler₃₉₃](#)

Description

Handlers and other objects can be run by a [Scheduler₉₇](#) if they provide a `run()` method and the methods defined below. The [Scheduler₉₇](#) uses this information to create a suitable context to execute the `run()` method.

6.1.1 Methods

```
public boolean addIfFeasible()
```

This method first performs a feasibility analysis with `this` added to the system. If the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should be considered in feasibility analysis until further notified. If the analysis showed that the system including `this` would not be feasible, this method does not admit `this` to the feasibility set.

If the object is already included in the feasibility set, do nothing.

Returns: True if inclusion of `this` in the feasibility set yields a feasible system, and false otherwise. If true is returned then `this` is known to be in the feasibility set. If false is returned `this` was not added to the feasibility set, but may already have been present.

Since: 1.0.1 Promoted to the Schedulable interface

public boolean addToFeasibility()

Inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should be considered in feasibility analysis until further notified.

If the object is already included in the feasibility set, do nothing.

Returns: True, if the resulting system is feasible. False, if not.

**public [javax.realtime.MemoryParameters₂₇₃](#)
getMemoryParameters()**

Gets a reference to the [MemoryParameters₂₇₃](#) object for this schedulable object.

Returns: A reference to the current [MemoryParameters₂₇₃](#) object.

**public [javax.realtime.ProcessingGroupParameters₁₄₃](#)
getProcessingGroupParameters()**

Gets a reference to the [ProcessingGroupParameters₁₄₃](#) object for this schedulable object.

Returns: A reference to the current [ProcessingGroupParameters₁₄₃](#) object.

**public [javax.realtime.ReleaseParameters₁₁₆](#)
getReleaseParameters()**

Gets a reference to the [ReleaseParameters₁₁₆](#) object for this schedulable object.

Returns: A reference to the current [ReleaseParameters₁₁₆](#) object.

public [javax.realtime.Scheduler₉₇](#) getScheduler()

Gets a reference to the [Scheduler₉₇](#) object for this schedulable object.

Returns: A reference to the associated [Scheduler₉₇](#) object.

**public [javax.realtime.SchedulingParameters₁₁₂](#)
getSchedulingParameters()**

Gets a reference to the [SchedulingParameters₁₁₂](#) object for this schedulable object.

Returns: A reference to the current [SchedulingParameters₁₁₂](#) object.

public boolean **removeFromFeasibility()**

Inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should *not* be considered in feasibility analysis until it is further notified.

Returns: True, if the removal was successful. False, if the schedulable object cannot be removed from the scheduler's feasibility set; e.g., the schedulable object is not part of the scheduler's feasibility set.

public boolean **setIfFeasible**(

[javax.realtime.ReleaseParameters₁₁₆](#) release,
[javax.realtime.MemoryParameters₂₇₃](#) memory)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of *this*. If the resulting system is feasible, this method replaces the current parameters of *this* with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

release - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

memory - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-

heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if `this` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or

[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

Since: 1.0.1 Promoted to the `Schedulable` interface.

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. If the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new

object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)₅₄](#) or

[RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

Since: 1.0.1 Promoted to the `Schedulable` interface.

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

release - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

group - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made.
False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

Since: 1.0.1 Promoted to the `Schedulable` interface.

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be

immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

Since: 1.0.1

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. If the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made.
False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-

heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or

[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

Since: 1.0.1

```
public void setMemoryParameters(
    javax.realtime.MemoryParameters273 memory)
```

Sets the memory parameters associated with this instance of `Schedulable`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the memory parameters of the existing schedulable objects, this may change the feasibility of the current system.

Parameters:

`memory` - A [MemoryParameters](#)₂₇₃ object which will become the memory parameters associated with this after the method call. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Throws:

`java.lang.IllegalArgumentException` - Thrown if `memory` is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and `memory` is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if the schedulable object cannot hold a reference to `memory`, or if `memory` cannot hold a reference to this schedulable object instance.

```
public boolean setMemoryParametersIfFeasible(
    javax.realtime.MemoryParameters273 memory)
```

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. If the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if `this` cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to `this`.

```
public void setProcessingGroupParameters(
    javax.realtime.ProcessingGroupParameters143 group)
```

Sets the [ProcessingGroupParameters](#)₁₄₃ of `this`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the processing group parameters of the existing schedulable objects, this may change the feasibility of the current system.

Parameters:

group - A [ProcessingGroupParameters₁₄₃](#) object which will take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

Throws:

[java.lang.IllegalArgumentException](#) - Thrown when **group** is not compatible with the scheduler for this schedulable object. Also thrown if this schedulable object is no-heap and **group** is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if **this** object cannot hold a reference to **group** or **group** cannot hold a reference to **this**.

```
public boolean setProcessingGroupParametersIfFeasible(
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of **this**. If the resulting system is feasible, this method replaces the current parameter of **this** with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

group - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

```
public void setReleaseParameters(
    javax.realtime.ReleaseParameters116 release)
```

Sets the release parameters associated with this instance of `Schedulable`.

Since this affects the constraints expressed in the release parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. The different properties of the release parameters may take effect at different times. See the documentation for the scheduler for details.

Parameters:

`release` - A `ReleaseParameters116` object which will become the release parameters associated with this after the method call, and take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `release` is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and `release` is located in heap memory.

`IllegalAssignmentError448` - Thrown if this object cannot hold a reference to `release` or `release` cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is

currently waiting for the next release in
[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or
[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public boolean setReleaseParametersIfFeasible(
    javax.realtime.ReleaseParameters116 release)
```

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in
[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or
[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler)
```

Sets the reference to the Scheduler object. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and scheduler.

Parameters:

scheduler - A reference to the scheduler that will manage execution of this schedulable object. Null is not a permissible value.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown when scheduler is null, or the schedulable object's existing parameter values are not compatible with scheduler. Also thrown if this schedulable object is no-heap and scheduler is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if the schedulable object cannot hold a reference to scheduler.

[java.lang.SecurityException](#) - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler,
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memoryParameters,
    javax.realtime.ProcessingGroupParameters143 group)
```

Sets the scheduler and associated parameter objects. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and scheduler.

Parameters:

scheduler - A reference to the scheduler that will manage the execution of this schedulable object. Null is not a permissible value.

scheduling - A reference to the [SchedulingParameters](#)₁₁₂ which will be associated with this. If null, the default value is governed by scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

release - A reference to the [ReleaseParameters](#)₁₁₆ which will be associated with this. If null, the default value is governed by

`scheduler` (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memoryParameters` - A reference to the [MemoryParameters₂₇₃](#) which will be associated with `this`. If null, the default value is governed by `scheduler` (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`group` - A reference to the [ProcessingGroupParameters₁₄₃](#) which will be associated with `this`. If null, the default value is governed by `scheduler` (a new object is created). (See [PriorityScheduler₁₀₃](#).)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable object is no-heap and `scheduler`, `scheduling release`, `memoryParameters`, or `group` is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if `this` object cannot hold references to all the parameter objects or the parameters cannot hold references to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

`java.lang.SecurityException` - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setSchedulingParameters(
    javax.realtime.SchedulingParameters112 scheduling)
```

Sets the scheduling parameters associated with this instance of `Schedulable`.

Since this affects the scheduling parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Parameters:

`scheduling` - A reference to the [SchedulingParameters₁₁₂](#) object. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduling` is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and `scheduling` is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this object cannot hold a reference to `scheduling` or `scheduling` cannot hold a reference to this.

```
public boolean setSchedulingParametersIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling)
```

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. If the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's

scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

6.2 Scheduler

Declaration

```
public abstract class Scheduler
```

Direct Known Subclasses: [PriorityScheduler](#)₁₀₃

Description

An instance of Scheduler manages the execution of schedulable objects and implements a feasibility algorithm. The feasibility algorithm determines if the known set of schedulable objects, given their particular execution ordering (or priority assignment), is a feasible system.

Subclasses of Scheduler are used for alternative scheduling policies and should define an `instance()` class method to return the default instance of the subclass. The name of the subclass should be descriptive of the policy, allowing applications to deduce the policy available for the scheduler obtained via [getDefaultScheduler\(\)](#)₉₈ (e.g., [EDFScheduler](#)).

6.2.1 Constructors

```
protected Scheduler()
```

Create an instance of Scheduler.

6.2.2 Methods

```
protected abstract boolean addToFeasibility(
    javax.realtime.Schedulable81 schedulable)
```

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable](#)₈₁ will be considered in the feasibility analysis of the associated [Scheduler](#)₉₇ until further notice. Whether the resulting system is feasible or not, the addition is completed.

If the object is already included in the feasibility set, do nothing.

Parameters:

`schedulable` - A reference to the given instance of `Schedulable81`

Returns: True, if the system is feasible after the addition. False, if not.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or if `schedulable` is not associated with this; that is `schedulable.getScheduler() != this`.

```
public abstract void fireSchedulable(
    javax.realtime.Schedulable81 schedulable)
```

Trigger the execution of a schedulable object (like an `AsyncEventHandler393`).

Parameters:

`schedulable` - The schedulable object to make active. If null, nothing happens.

Throws:

`java.lang.UnsupportedOperationException` - Thrown if the scheduler cannot release `schedulable` for execution.

```
public static javax.realtime.Scheduler97
    getDefaultScheduler()
```

Gets a reference to the default scheduler.

Returns: A reference to the default scheduler.

```
public abstract java.lang.String getPolicyName()
```

Gets a string representing the policy of this. The string value need not be interned, but it must be created in a memory area that does not cause an illegal assignment error if stored in the current allocation context and does not cause a `MemoryAccessError450` when accessed.

Returns: A `java.lang.String` object which is the name of the scheduling policy used by this.

public abstract boolean **isFeasible()**

Queries the system about the feasibility of the system currently being considered. The definitions of “feasible” and “system” are the responsibility of the feasibility algorithm of the actual Scheduler subclass.

Returns: True, if the system is feasible. False, if not.

protected abstract boolean **removeFromFeasibility()** [javax.realtime.Schedulable₈₁](#) schedulable)

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable₈₁](#) should no longer be considered in the feasibility analysis of the associated [Scheduler₉₇](#). Whether the resulting system is feasible or not, the removal is completed.

Parameters:

schedulable - A reference to the given instance of [Schedulable₈₁](#)

Returns: True, if the removal was successful. False, if the schedulable object cannot be removed from the scheduler’s feasibility set; e.g., the schedulable object is not part of the scheduler’s feasibility set.

Throws:

java.lang.IllegalArgumentException - Thrown if schedulable is null.

public static void **setDefaultScheduler()** [javax.realtime.Scheduler₉₇](#) scheduler)

Sets the default scheduler. This is the scheduler given to instances of schedulable objects when they are constructed by a Java thread. The default scheduler is set to the required [PriorityScheduler₁₀₃](#) at startup.

Parameters:

scheduler - The Scheduler that becomes the default scheduler assigned to new schedulable objects created by Java threads. If null nothing happens.

Throws:

java.lang.SecurityException - Thrown if the caller is not permitted to set the default scheduler.

```
public abstract boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

`schedulable` - The schedulable object for which the changes are proposed.

`release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memory` - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

`IllegalAssignmentError448` - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

```
public abstract boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

- `schedulable` - The schedulable object for which the changes are proposed.
- `release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `memory` - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `group` - The proposed processing group parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made.
False, if the resulting system is not feasible and no changes are made.

Throws:

- `java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.
- `IllegalAssignmentError448` - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.
- `java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently

waiting for the next release in
[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or
[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public abstract boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Parameters:

- `schedulable` - The schedulable object for which the changes are proposed.
- `scheduling` - The proposed scheduling parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `memory` - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)
- `group` - The proposed processing group parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

- `java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with

this scheduler, or the proposed parameters are not compatible with this scheduler.

`IllegalAssignmentError`₄₄₈ - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently waiting for the next release in

`RealtimeThread.waitForNextPeriod()`₅₄ or

`RealtimeThread.waitForNextPeriodInterruptible()`₅₄.

6.3 PriorityScheduler

Declaration

public class **PriorityScheduler** extends `Scheduler`₉₇

Description

Class which represents the required (by the RTSJ) priority-based scheduler. The default instance is the base scheduler which does fixed priority, preemptive scheduling.

This scheduler, like all schedulers, governs the default values for scheduling-related parameters in its client schedulable objects. The defaults are as follows:

Attribute	Default Value
<i>Priority parameters</i>	
Priority	norm priority

Note that the system contains one instance of the `PriorityScheduler` which is the system's base scheduler and is returned by `PriorityScheduler.instance()`. It may, however, contain instances of subclasses of `PriorityScheduler` and even additional instances of `PriorityScheduler` itself created through this class' protected constructor. The instance returned by the `instance()`₁₀₇ method is the *base scheduler* and is returned by `Scheduler.getDefaultScheduler()`₉₈ unless the default scheduler is reset with

`Scheduler.setDefaultScheduler(Scheduler)`₉₉.

6.3.1 Fields

```
public static final int MAX_PRIORITY
```

Deprecated. 1.0.1 Use the [getMaxPriority\(\)](#)₁₀₅ method instead.

The maximum priority value used by the implementation.

```
public static final int MIN_PRIORITY
```

Deprecated. 1.0.1 Use the [getMinPriority\(\)](#)₁₀₅ method instead.

The minimum priority value used by the implementation.

6.3.2 Constructors

```
protected PriorityScheduler()
```

Construct an instance of `PriorityScheduler`. Applications will likely not need any instance other than the default instance.

6.3.3 Methods

```
protected boolean addToFeasibility(
    javax.realtime.Schedulable81 schedulable)
```

Description copied from class: [javax.realtime.Scheduler](#)₉₇

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable](#)₈₁ will be considered in the feasibility analysis of the associated [Scheduler](#)₉₇ until further notice. Whether the resulting system is feasible or not, the addition is completed.

If the object is already included in the feasibility set, do nothing.

Overrides: [addToFeasibility](#)₉₇ in class [Scheduler](#)₉₇

Parameters:

`schedulable` - A reference to the given instance of [Schedulable](#)₈₁

Returns: True, if the system is feasible after the addition. False, if not.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or if `schedulable` is not associated with `this`; that is `schedulable.getScheduler() != this`.

```
public void fireSchedulable(
    javax.realtime.Schedulable81 schedulable)
```

Description copied from class: [javax.realtime.Scheduler₉₇](#)

Trigger the execution of a schedulable object (like an [AsyncEventHandler₃₉₃](#)).

Overrides: [fireSchedulable₉₈](#) in class [Scheduler₉₇](#)

Parameters:

schedulable - The schedulable object to make active. If null, nothing happens.

Throws:

[java.lang.UnsupportedOperationException](#) - Thrown in all cases by the [PriorityScheduler](#)

```
public int getMaxPriority()
```

Gets the maximum priority available for a schedulable object managed by this scheduler.

Returns: The value of the maximum priority.

```
public static int getMaxPriority(java.lang.Thread thread)
```

Gets the maximum priority for the given thread. If the given thread is a real-time thread that is scheduled by an instance of [PriorityScheduler](#), then the maximum priority for that scheduler is returned. If the given thread is a Java thread then the maximum priority of its thread group is returned. Otherwise an exception is thrown.

Parameters:

thread - An instance of [java.lang.Thread](#). If null, the maximum priority of this scheduler is returned.

Returns: The maximum priority for thread

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if thread is a real-time thread that is not scheduled by an instance of [PriorityScheduler](#).

```
public int getMinPriority()
```

Gets the minimum priority available for a schedulable object managed by this scheduler.

Returns: The minimum priority used by this scheduler.

```
public static int getMinPriority(java.lang.Thread thread)
```

Gets the minimum priority for the given thread. If the given thread is a real-time thread that is scheduled by an instance of `PriorityScheduler`, then the minimum priority for that scheduler is returned. If the given thread is a Java thread then `Thread.MIN_PRIORITY` is returned. Otherwise an exception is thrown.

Parameters:

`thread` - An instance of `java.lang.Thread`. If null, the minimum priority of this scheduler is returned.

Returns: The minimum priority for thread

Throws:

`java.lang.IllegalArgumentException` - Thrown if thread is a real-time thread that is not scheduled by an instance of `PriorityScheduler`.

```
public int getNormPriority()
```

Gets the normal priority available for a schedulable object managed by this scheduler.

Returns: The value of the normal priority.

```
public static int getNormPriority(java.lang.Thread thread)
```

Gets the “norm” priority for the given thread. If the given thread is a real-time thread that is scheduled by an instance of `PriorityScheduler`, then the norm priority for that scheduler is returned. If the given thread is a Java thread then `Thread.NORM_PRIORITY` is returned. Otherwise an exception is thrown.

Parameters:

`thread` - An instance of `java.lang.Thread`. If null, the norm priority for this scheduler is returned.

Returns: The norm priority for thread

Throws:

`java.lang.IllegalArgumentException` - Thrown if thread is a real-time thread that is not scheduled by an instance3 of `PriorityScheduler`.

```
public java.lang.String getPolicyName()
```

Gets the policy name of this.

Overrides: [getPolicyName₉₈](#) in class [Scheduler₉₇](#)

Returns: The policy name (Fixed Priority) as a string.

```
public static javax.realtime.PriorityScheduler103 instance()
```

Return a reference to the distinguished instance of [PriorityScheduler](#) which is the system's base scheduler.

Returns: A reference to the distinguished instance [PriorityScheduler](#).

```
public boolean isFeasible()
```

Queries this scheduler about the feasibility of the set of schedulable objects currently in the feasibility set.

Implementation Notes:

The default feasibility test for the [PriorityScheduler](#) considers a set of schedulable objects with bounded resource requirements, to always be feasible. This covers all schedulable objects with release parameters of types [PeriodicParameters₁₂₂](#) and [SporadicParameters₁₃₆](#).

If any schedulable object within the feasibility set has release parameters of the exact type [AperiodicParameters₁₂₉](#) (not a subclass thereof), then the feasibility set is not feasible, as aperiodic release characteristics require unbounded resources. In that case, this method will return `false` and all methods in the `setIfFeasible` family of methods will also return `false`. Consequently, any call to a `setIfFeasible` method that passes a schedulable object which has release parameters of type [AperiodicParameters₁₂₉](#), or passes proposed release parameters of type [AperiodicParameters₁₂₉](#), will return `false`. The only time a `setIfFeasible` method can return `true`, when there exists in the feasibility set a schedulable object with release parameters of type [AperiodicParameters₁₂₉](#), is when the method will change those release parameters to not be [AperiodicParameters₁₂₉](#).

Implementations may provide a feasibility test other than the default test just described. In which case the details of that test should be documented here in place of this description of the default implementation.

Overrides: [isFeasible₉₉](#) in class [Scheduler₉₇](#)

```
protected boolean removeFromFeasibility(
    javax.realtime.Schedulable81 schedulable)
```

Description copied from class: [javax.realtime.Scheduler₉₇](#)

Inform this scheduler and cooperating facilities that the resource demands of the given instance of [Schedulable₈₁](#) should no longer be considered in the feasibility analysis of the associated [Scheduler₉₇](#). Whether the resulting system is feasible or not, the removal is completed.

Overrides: [removeFromFeasibility₉₉](#) in class [Scheduler₉₇](#)

Parameters:

`schedulable` - A reference to the given instance of [Schedulable₈₁](#)

Returns: True, if the removal was successful. False, if the schedulable object cannot be removed from the scheduler's feasibility set; e.g., the schedulable object is not part of the scheduler's feasibility set.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null.

```
public boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

Description copied from class: [javax.realtime.Scheduler₉₇](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Overrides: [setIfFeasible₁₀₀](#) in class [Scheduler₉₇](#)

Parameters:

`schedulable` - The schedulable object for which the changes are proposed.

`release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

memory - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

[IllegalAssignmentError₄₄₈](#) - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)₅₄](#) or

[RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

```
public boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from class: [javax.realtime.Scheduler₉₇](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Overrides: [setIfFeasible₁₀₁](#) in class [Scheduler₉₇](#)

Parameters:

`schedulable` - The schedulable object for which the changes are proposed.

`release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

`memory` - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

`group` - The proposed processing group parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#) .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.

[IllegalAssignmentError₄₄₈](#) - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

```
public boolean setIfFeasible(
    javax.realtime.Schedulable81 schedulable,
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from class: [javax.realtime.Scheduler₉₇](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `schedulable`. If the resulting system is feasible, this method replaces the current parameters of `schedulable` with the proposed ones.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Overrides: [setIfFeasible₁₀₂](#) in class [Scheduler₉₇](#)

Parameters:

- `schedulable` - The schedulable object for which the changes are proposed.
- `scheduling` - The proposed scheduling parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `release` - The proposed release parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `memory` - The proposed memory parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `group` - The proposed processing group parameters. If null, the default value of this scheduler is used (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

- `java.lang.IllegalArgumentException` - Thrown if `schedulable` is null, or `schedulable` is not associated with this scheduler, or the proposed parameters are not compatible with this scheduler.
- `IllegalAssignmentError448` - Thrown if `schedulable` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `schedulable`.
- `java.lang.IllegalThreadStateException` - Thrown if the new release parameters change `schedulable` from periodic scheduling to some other protocol and `schedulable` is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

6.4 SchedulingParameters

Declaration

```
public abstract class SchedulingParameters implements
    java.lang.Cloneable
```

All Implemented Interfaces: `java.lang.Cloneable`

Direct Known Subclasses: [PriorityParameters₁₁₃](#)

Description

Subclasses of `SchedulingParameters` ([PriorityParameters₁₁₃](#), [ImportanceParameters₁₁₄](#), and any others defined for particular schedulers) provide the parameters to be used by the [Scheduler₉₇](#). Changes to the values in a parameters object affects the scheduling behavior of all the [Schedulable₈₁](#) objects to which it is bound.

Caution: Subclasses of this class are explicitly unsafe in multithreaded situations when they are being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

6.4.1 Constructors

```
protected SchedulingParameters()
```

Create a new instance of `SchedulingParameters`.

Since: 1.0.1

6.4.2 Methods

```
public java.lang.Object clone()
```

Return a clone of `this`. This method should behave effectively as if it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.

- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Overrides: `clone` in class `Object`

Since: 1.0.1

6.5 PriorityParameters

Declaration

`public class PriorityParameters extends SchedulingParameters112`

All Implemented Interfaces: `java.lang.Cloneable`

Direct Known Subclasses: [ImportanceParameters](#)₁₁₄

Description

Instances of this class should be assigned to schedulable objects that are managed by schedulers which use a single integer to determine execution order. The base scheduler required by this specification and represented by the class [PriorityScheduler](#)₁₀₃ is such a scheduler.

6.5.1 Constructors

```
public PriorityParameters(int priority)
```

Create an instance of [PriorityParameters](#)₁₁₃ with the given priority.

Parameters:

`priority` - The priority assigned to schedulable objects that use this parameter instance.

6.5.2 Methods

```
public int getPriority()
```

Gets the priority value.

Returns: The priority.

```
public void setPriority(int priority)
```

Set the priority value. If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

[RealtimeThread.setSchedulingParameters\(SchedulingParameters\)](#)₅₀) the base priority of those schedulable objects is altered as specified by each schedulable object's scheduler.

Parameters:

priority - The value to which priority is set.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the given priority value is incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

```
public java.lang.String toString()
```

Converts the priority value to a string.

Overrides: `toString` in class `Object`

Returns: A string representing the value of priority.

6.6 ImportanceParameters

Declaration

```
public class ImportanceParameters extends PriorityParameters113
```

All Implemented Interfaces: `java.lang.Cloneable`

Description

Importance is an additional scheduling metric that may be used by some priority-based scheduling algorithms during overload conditions to differentiate execution order among threads of the same priority.

In some real-time systems an external physical process determines the period of many threads. If rate-monotonic priority assignment is used to assign priorities many of the threads in the system may have the same priority because their periods are the same. However, it is conceivable that some threads may be more important than others and in an overload situation importance can help the scheduler decide which threads

to execute first. The base scheduling algorithm represented by [PriorityScheduler₁₀₃](#) must not consider importance.

6.6.1 Constructors

```
public ImportanceParameters(int priority, int importance)
```

Create an instance of ImportanceParameters.

Parameters:

priority - The priority value assigned to schedulable objects that use this parameter instance. This value is used in place of the value passed to `java.lang.Thread.setPriority(int)`.

importance - The importance value assigned to schedulable objects that use this parameter instance.

6.6.2 Methods

```
public int getImportance()
```

Gets the importance value.

Returns: The value of importance for the associated instances of [Schedulable₈₁](#).

```
public void setImportance(int importance)
```

Set the importance value. If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

[RealtimeThread.setSchedulingParameters\(SchedulingParameters\)₅₀](#)) the importance of those schedulable objects is altered at a

moment controlled by the schedulers for the respective schedulable objects.

Parameters:

importance - The value to which importance is set.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the given importance value is incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

```
public java.lang.String toString()
```

Print the value of the priority and importance values of the associated instance of [Schedulable₈₁](#)

Overrides: [toString₁₁₄](#) in class [PriorityParameters₁₁₃](#)

6.7 ReleaseParameters

Declaration

```
public abstract class ReleaseParameters implements
    java.lang.Cloneable
```

All Implemented Interfaces: [java.lang.Cloneable](#)

Direct Known Subclasses: [AperiodicParameters₁₂₉](#), [PeriodicParameters₁₂₂](#)

Description

The top-level class for release characteristics of schedulable objects. When a reference to a `ReleaseParameters` object is given as a parameter to a constructor, the `ReleaseParameters` object becomes bound to the object being created. Changes to the values in the `ReleaseParameters` object affect the constructed object. If given to more than one constructor, then changes to the values in the `ReleaseParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many. Only changes to an `ReleaseParameters` object caused by methods on that object cause the change to propagate to all schedulable objects using the object. For instance, invoking `setDeadline` on a `ReleaseParameters` instance will make the change, and then notify that the scheduler that the object has been changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the `RelativeTime` object that is the deadline for this object may change the time value but it does not pass the new time value to the scheduler at that time. Even though the changed time value is referenced by `ReleaseParameters` objects, it will not change the behavior of the SOs that use the parameter object until a setter method on the `ReleaseParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

Release parameters use [HighResolutionTime₃₁₄](#) values for cost, and deadline. Since the times are expressed as a [HighResolutionTime₃₁₄](#) values, these values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity the timers measure depend on the clock associated with each time value.

The implementation must use modified copy semantics for each [HighResolutionTime₃₁₄](#) parameter value. The value of each time object should be

treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by `getCost()` must be the same object passed in by `setCost()`, but any changes made to the time value of the cost must not take effect in the associated `ReleaseParameters` instance unless they are passed to the parameter object again, e.g. with a new invocation of `setCost`.

Attribute	Default Value
cost	new RelativeTime(0,0)
deadline	no default
overrunHandler	None
missHandler	None

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Caution: The cost parameter time should be considered to be measured against the target platform.

Note: Cost measurement and enforcement is an optional facility for implementations of the RTSJ.

6.7.1 Constructors

protected `ReleaseParameters()`

Create a new instance of `ReleaseParameters`. This constructor creates a default `ReleaseParameters` object, i.e., it is equivalent to `ReleaseParameters(null, null, null, null)`.

protected `ReleaseParameters(`

```

javax.realtime.RelativeTime332 cost,
javax.realtime.RelativeTime332 deadline,
javax.realtime.AsyncEventHandler393 overrunHandler,
javax.realtime.AsyncEventHandler393 missHandler)

```

Create a new instance of `ReleaseParameters` with the given parameter values.

Parameters:

- `cost` - Processing time units per release. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives per release. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds `cost`. If null, the default value is a new instance of `RelativeTime(0, 0)`.
- `deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. There is no default for `deadline` in this class. The default must be determined by the subclasses.
- `overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds `cost`. In the minimum implementation `overrunHandler` is ignored. If null, no application event handler is executed on `cost` overrun.
- `missHandler` - 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` - Thrown if the time value of `cost` is less than zero, or the time value of `deadline` is less than or equal to zero.

`IllegalAssignmentError448` - Thrown if `cost`, `deadline`, `overrunHandler`, or `missHandler` cannot be stored in this.

See Also: [RealtimeThread.waitForNextPeriod\(\)₅₄](#)

6.7.2 Methods

```
public java.lang.Object clone()
```

Return a clone of `this`. This method should behave effectively as if it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Overrides: `clone` in class `Object`

Since: 1.0.1

```
public javax.realtime.RelativeTime332 getCost()
```

Gets a reference to the cost.

Returns: A reference to cost.

```
public javax.realtime.AsyncEventHandler393
getCostOverrunHandler()
```

Gets a reference to the cost overrun handler.

Returns: A reference to the associated cost overrun handler.

```
public javax.realtime.RelativeTime332 getDeadline()
```

Gets a reference to the deadline.

Returns: A reference to deadline.

```
public javax.realtime.AsyncEventHandler393
getDeadlineMissHandler()
```

Gets a reference to the deadline miss handler.

Returns: A reference to the deadline miss handler.

```
public void setCost(javax.realtime.RelativeTime332 cost)
```

Sets the cost value.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

[RealtimeThread.setReleaseParameters\(ReleaseParameters\)46](#))

the cost of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Parameters:

`cost` - Processing time units per release. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives per release. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds `cost`. If null, the default value is a new instance of `RelativeTime(0,0)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `cost` is less than zero.

`IllegalAssignmentError448` - Thrown if `cost` cannot be stored in this.

```
public void setCostOverrunHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Sets the cost overrun handler.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

`RealtimeThread.setReleaseParameters(ReleaseParameters)46`) the cost overrun handler of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Parameters:

`handler` - This handler is invoked if an invocation of the schedulable object attempts to exceed `cost` time units in a release. A null value of `handler` signifies that no cost overrun handler should be used.

Throws:

`IllegalAssignmentError448` - Thrown if `handler` cannot be stored in this.

```
public void setDeadline(
    javax.realtime.RelativeTime332 deadline)
```

Sets the deadline value.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method

such as

`RealtimeThread.setReleaseParameters(ReleaseParameters)46`)

the deadline of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Parameters:

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. The default value of the deadline must be controlled by the classes that extend `ReleaseParameters`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `deadline` is null, the time value of `deadline` is less than or equal to zero, or if the new value of this deadline is incompatible with the scheduler for any associated schedulable object.

`IllegalAssignmentError448` - Thrown if `deadline` cannot be stored in this.

```
public void setDeadlineMissHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Sets the deadline miss handler.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

`RealtimeThread.setReleaseParameters(ReleaseParameters)46`)

the deadline miss handler of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Parameters:

`handler` - This handler is invoked if any release of the schedulable object fails to complete before the deadline passes. Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate. A null value of `handler` signifies that no deadline miss handler should be used.

Throws:

`IllegalAssignmentError448` - Thrown if `handler` cannot be stored in this.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the new cost, and deadline as replacements for the matching attributes of all schedulable objects associated with this release parameters object. If the resulting system is feasible, the method replaces the current scheduling characteristics of this release parameters object with the new scheduling characteristics. The change in the release characteristics, including the timing of the change, of any associated schedulable objects will take place under the control of their schedulers.

Parameters:

`cost` - The proposed cost. Equivalent to `RelativeTime(0, 0)` if null. (A new instance of `RelativeTime332` is created in the memory area containing this `ReleaseParameters` instance). If null, the default value is a new instance of `RelativeTime(0, 0)`.

`deadline` - The proposed deadline. If null, the default value is new instance of `RelativeTime(period)`.

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `cost` is less than zero, or the time value of `deadline` is less than or equal to zero.

`IllegalAssignmentError448` - Thrown if `cost` or `deadline` cannot be stored in this.

6.8 PeriodicParameters

Declaration

```
public class PeriodicParameters extends ReleaseParameters116
```

All Implemented Interfaces: `java.lang.Cloneable`

Description

This release parameter indicates that the schedulable object is released on a regular basis. For an `AsyncEventHandler393`, this means that the handler is either released

by a periodic timer, or the associated event occurs periodically. For a [RealtimeThread₂₉](#), this means that the [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#) method will unblock the associated real-time thread at the start of each period.

When a reference to a `PeriodicParameters` object is given as a parameter to a schedulable object's constructor or passed as an argument to one of the schedulable object's setter methods, the `PeriodicParameters` object becomes the release parameters object bound to that schedulable object. Changes to the values in the `PeriodicParameters` object affect that schedulable object. If bound to more than one schedulable object then changes to the values in the `PeriodicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

Only changes to a `PeriodicParameters` object caused by methods on that object cause the change to propagate to all schedulable objects using the object. For instance, calling `setCost` on an `PeriodicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the SOs that use the parameter object until a setter method on the `PeriodicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

Periodic parameters use [HighResolutionTime₃₁₄](#) values for period and start time. Since these times are expressed as a [HighResolutionTime₃₁₄](#) values, these values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity the timers measure depend on the clock associated with each time value.

The implementation must use modified copy semantics for each [HighResolutionTime₃₁₄](#) parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by `getCost()` must be the same object passed in by `setCost()`, but any changes made to the time value of the cost must not take effect in the associated `PeriodicParameters` instance unless they are passed to the parameter object again, e.g. with a new invocation of `setCost`.

Periodic release parameters are strictly informational when they are applied to async event handlers. They must be used for any feasibility analysis, but release of the async event handler is not entirely controlled by the scheduler.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Caution: An implementation is not required to ensure that each `AsyncEventHandler` with periodic parameters is released periodically.

Attribute	Default Value
start	new RelativeTime(0,0)
period	No default. A value must be supplied
cost	new RelativeTime(0,0)
deadline	new RelativeTime(period)
overrunHandler	None
missHandler	None

6.8.1 Constructors

```
public PeriodicParameters(
    javax.realtime.HighResolutionTime314 start,
    javax.realtime.RelativeTime332 period)
```

Create a `PeriodicParameters` object. This constructor has the same effect as invoking `PeriodicParameters(start, period, null, null, null, null)`

Parameters:

`start` - Time at which the first release begins (i.e. the real-time thread becomes eligible for execution.) If a `RelativeTime`, this time is relative to the first time the thread becomes activated (that is, when `start()` is called). If an `AbsoluteTime`, then the first release is the maximum of the start parameter and the time of the call to the associated `RealtimeThread.start()` method. If null, the default value is a new instance of `RelativeTime(0, 0)`.

`period` - The period is the interval between successive unblocks of the `RealtimeThread.waitForNextPeriod()`₅₄ and `RealtimeThread.waitForNextPeriodInterruptible()`₅₄ methods. There is no default value. If `period` is null an exception is thrown.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the period is null or its time value is not greater than zero.

`IllegalAssignmentError`₄₄₈ - Thrown if start or period cannot be stored in this.

Since: 1.0.1

public PeriodicParameters(

```

    javax.realtime.HighResolutionTime314 start,
    javax.realtime.RelativeTime332 period,
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline,
    javax.realtime.AsyncEventHandler393 overrunHandler,
    javax.realtime.AsyncEventHandler393 missHandler)

```

Create a `PeriodicParameters` object.

Parameters:

`start` - Time at which the first release begins (i.e. the real-time thread becomes eligible for execution.) If a `RelativeTime`, this time is relative to the first time the thread becomes activated (that is, when `start()` is called). If an `AbsoluteTime`, then the first release is the maximum of the start parameter and the time of the call to the associated `RealtimeThread.start()` method. If null, the default value is a new instance of `RelativeTime(0,0)`.

`period` - The period is the interval between successive unblocks of the `RealtimeThread.waitForNextPeriod()`₅₄ and `RealtimeThread.waitForNextPeriodInterruptible()`₅₄ methods. There is no default value. If `period` is null an exception is thrown.

`cost` - Processing time per release. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives per release. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds or will exceed cost time units in a release. If null, the default value is a new instance of `RelativeTime(0,0)`.

`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 new instance of `RelativeTime(period)`.

`overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds cost in the given release. Implementations may ignore this parameter. If null, the default value no overrun handler.

`missHandler` - 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` - Thrown 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.

`IllegalAssignmentError`₄₄₈ - Thrown if start, period, cost, deadline, `overrunHandler` or `missHandler` cannot be stored in this.

```
public PeriodicParameters(
    javax.realtime.RelativeTime332 period)
```

Create a `PeriodicParameters` object. This constructor has the same effect as invoking `PeriodicParameters(null, period, null, null, null, null)`

Parameters:

`period` - The period is the interval between successive unblocks of the `RealtimeThread.waitForNextPeriod()`₅₄ and `RealtimeThread.waitForNextPeriodInterruptible()`₅₄ methods. There is no default value. If `period` is null an exception is thrown.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the period is null or its time value is not greater than zero.

`IllegalAssignmentError`₄₄₈ - Thrown if period cannot be stored in this.

Since: 1.0.1

6.8.2 Methods

```
public javax.realtime.RelativeTime332 getPeriod()
```

Gets the period.

Returns: The current value in period.

```
public javax.realtime.HighResolutionTime314 getStart()
```

Gets the start time.

Returns: The current value in start. This is the value that was specified in the constructor or by `setStart()`, not the actual absolute time corresponding to the start of one of the schedulable objects associated with this `PeriodicParameters` object.

```
public void setDeadline(  
    javax.realtime.RelativeTime332 deadline)
```

Sets the deadline value.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

[RealtimeThread.setReleaseParameters\(ReleaseParameters\)](#)₄₆)

the deadline of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Overrides: [setDeadline](#)₁₂₀ in class [ReleaseParameters](#)₁₁₆

Parameters:

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. If `deadline` is null, the deadline is set to a new instance of `RelativeTime` equal to period.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `deadline` is less than or equal to zero, or if the new value of this deadline is incompatible with the scheduler for any associated schedulable object.

`IllegalAssignmentError`₄₄₈ - Thrown if `deadline` cannot be stored in this.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 period,
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the new period, cost and deadline attributes as replacements for the matching attributes of `this`. If the resulting system is feasible the method replaces the current attributes of `this`. If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or `set` with a method such as

[RealtimeThread.setReleaseParameters\(ReleaseParameters\)₄₆](#)) the parameters of those schedulable objects are altered as specified by each schedulable object's respective scheduler.

Parameters:

`period` - The proposed period. There is no default value. If `period` is null an exception is thrown.

`cost` - The proposed cost. If null, the default value is a new instance of `RelativeTime(0, 0)`.

`deadline` - The proposed deadline. If null, the default value is new instance of `RelativeTime(period)`.

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown 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. Also thrown if the values are incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

[IllegalAssignmentError₄₄₈](#) - Thrown if `period`, `cost` or `deadline` cannot be stored in `this`.

```
public void setPeriod(javax.realtime.RelativeTime332 period)
```

Sets the period.

Parameters:

`period` - The value to which `period` is set.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the given period is null or its time value is not greater than zero. Also thrown if period is incompatible with the scheduler for any associated schedulable object.

`IllegalAssignmentError448` - Thrown if period cannot be stored in this.

```
public void setStart(
    javax.realtime.HighResolutionTime314 start)
```

Sets the start time.

The effect of changing the start time for any schedulable objects associated with this parameter object is determined by the scheduler associated with each schedulable object.

Note: An instance of `PeriodicParameters` may be shared by several schedulable objects. A change to the start time may take effect on a subset of these schedulable objects. That leaves the start time returned by `getStart` unreliable as a way to determine the start time of a schedulable object.

Parameters:

`start` - The new start time. If null, the default value is a new instance of `RelativeTime(0,0)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the given start time is incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

`IllegalAssignmentError448` - Thrown if start cannot be stored in this.

6.9 AperiodicParameters

Declaration

```
public class AperiodicParameters extends ReleaseParameters116
```

All Implemented Interfaces: `java.lang.Cloneable`

Direct Known Subclasses: `SporadicParameters136`

Description

This release parameter object characterizes a schedulable object that may be released at any time.

When a reference to an `AperiodicParameters` object is given as a parameter to a schedulable object's constructor or passed as an argument to one of the schedulable object's setter methods, the `AperiodicParameters` object becomes the release parameters object bound to that schedulable object. Changes to the values in the `AperiodicParameters` object affect that schedulable object. If bound to more than one schedulable object then changes to the values in the `AperiodicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

Only changes to an `AperiodicParameters` object caused by methods on that object cause the change to propagate to all schedulable objects using the object. For instance, calling `setCost` on an `AperiodicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every `SO` that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the `SOs` that use the parameter object until a setter method on the `AperiodicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an `SO`.

The implementation must use modified copy semantics for each `HighResolutionTime314` parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by `getCost()` must be the same object passed in by `setCost()`, but any changes made to the time value of the cost must not take effect in the associated `AperiodicParameters` instance unless they are passed to the parameter object again, e.g. with a new invocation of `setCost`.

Correct initiation of the deadline miss and cost overrun handlers requires that the underlying system know the arrival time of each sporadic task. For an instance of `RealtimeThread29` the arrival time is the time at which the `start()` is invoked. For other instances of `Schedulable81` required behaviors may require the implementation to behave effectively as if it maintained a queue of arrival times.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Attribute	Value
cost	new RelativeTime(0,0)
deadline	new RelativeTime(Long.MAX_VALUE, 999999)
overrunHandler	None
missHandler	None
Arrival time queue size	0
Queue overflow policy	SAVE

Correct initiation of the deadline miss and cost overrun handlers requires that the underlying system know the arrival time of each aperiodic task. For an instance of [RealtimeThread₂₉](#) the arrival time is the time at which the `start()` is invoked. For other instances of [Schedulable₈₁](#) required behaviors may require the implementation to behave effectively as if it maintained a queue of arrival times.

6.9.1 Fields

```
public static final java.lang.String
    arrivalTimeQueueOverflowExcept
```

Represents the “EXCEPT” policy for dealing with arrival time queue overflow. Under this policy, if an arrival occurs and its time should be queued but the queue already holds a number of times equal to the initial queue length defined by this then the `fire()` method shall throw a [ArrivalTimeQueueOverflowException₄₄₆](#). Any other associated semantics are governed by the schedulers for the schedulable objects using these aperiodic parameters. If the arrival is a result of a happening to which the instance of [AsyncEventHandler₃₉₃](#) is bound then the arrival time is ignored.

Since: 1.0.1 Moved here from `SporadicParameters`.

```
public static final java.lang.String
    arrivalTimeQueueOverflowIgnore
```

Represents the “IGNORE” policy for dealing with arrival time queue overflow. Under this policy, if an arrival occurs and its time should be queued, but the queue already holds a number of times equal to the initial queue

length defined by `this` then the arrival is ignored. Any other associated semantics are governed by the schedulers for the schedulable objects using these aperiodic parameters.

Since: 1.0.1 Moved here from `SporadicParameters`.

```
public static final java.lang.String
    arrivalTimeQueueOverflowReplace
```

Represents the “REPLACE” policy for dealing with arrival time queue overflow. Under this policy if an arrival occurs and should be queued but the queue already holds a number of times equal to the initial queue length defined by `this` then the information for this arrival replaces a previous arrival. Any other associated semantics are governed by the schedulers for the schedulable objects using these aperiodic parameters.

Since: 1.0.1 Moved here from `SporadicParameters`.

```
public static final java.lang.String
    arrivalTimeQueueOverflowSave
```

Represents the “SAVE” policy for dealing with arrival time queue overflow. Under this policy if an arrival occurs and should be queued but the queue is full, then the queue is lengthened and the arrival time is saved. Any other associated semantics are governed by the schedulers for the schedulable objects using these aperiodic parameters.

This policy does not update the “initial queue length” as it alters the actual queue length. Since the SAVE policy grows the arrival time queue as necessary, for the SAVE policy the initial queue length is only an optimization.

Since: 1.0.1 Moved here from `SporadicParameters`.

6.9.2 Constructors

```
public AperiodicParameters()
```

Create an `AperiodicParameters` object. This constructor is equivalent to: `AperiodicParameters(null, null, null, null)`

Since: 1.0.1

```
public AperiodicParameters(
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline,
    javax.realtime.AsyncEventHandler393 overrunHandler,
    javax.realtime.AsyncEventHandler393 missHandler)
```

Create an AperiodicParameters object.

Parameters:

`cost` - Processing time per invocation. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds `cost`. If null, the default value is a new instance of `RelativeTime(0,0)`.

`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)`.

`overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds `cost`. Not required for minimum implementation. If null, the default value is no overrun handler.

`missHandler` - 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 is no miss handler.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `cost` is less than zero, or the time value of `deadline` is less than or equal to zero.

`IllegalAssignmentError448` - Thrown if `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in this.

6.9.3 Methods

```
public java.lang.String
    getArrivalTimeQueueOverflowBehavior()
```

Gets the behavior of the arrival time queue in the event of an overflow.

Returns: The behavior of the arrival time queue as a string.

Since: 1.0.1 Moved from `SporadicParameters`

```
public int getInitialArrivalTimeQueueLength()
```

Gets the initial number of elements the arrival time queue can hold. This returns the initial queue length currently associated with this parameter object. If the overflow policy is `SAVE` the initial queue length may not be related to the current queue lengths of schedulable objects associated with this parameter object.

Returns: The initial length of the queue.

Since: 1.0.1 Moved here from `SporadicParameters`.

```
public void setArrivalTimeQueueOverflowBehavior(
    java.lang.String behavior)
```

Sets the behavior of the arrival time queue in the case where the insertion of a new element would make the queue size greater than the initial size given in this.

Values of `behavior` are compared using reference equality (`==`) not value equality (`equals()`).

Parameters:

`behavior` - A string representing the behavior.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `behavior` is not one of the final queue overflow behavior values defined in this class.

Since: 1.0.1 Moved here from `SporadicParameters`.

```
public void setDeadline(
    javax.realtime.RelativeTime332 deadline)
```

Sets the deadline value.

If this parameter object is associated with any schedulable object (by being passed through the schedulable object's constructor or set with a method such as

[RealtimeThread.setReleaseParameters\(ReleaseParameters\)](#)₄₆)

the deadline of those schedulable objects is altered as specified by each schedulable object's respective scheduler.

Overrides: [setDeadline](#)₁₂₀ in class [ReleaseParameters](#)₁₁₆

Parameters:

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. If `deadline` is null, the deadline is set to a new instance of `RelativeTime(Long.MAX_VALUE, 999999)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `deadline` is less than or equal to zero, or if the new value of this deadline is incompatible with the scheduler for any associated schedulable object.

`IllegalAssignmentError`₄₄₈ - Thrown if `deadline` cannot be stored in this.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the new `cost`, and `deadline` as replacements for the matching attributes of this. If the resulting system is feasible, the method replaces the current scheduling characteristics, of this with the new scheduling characteristics.

Overrides: [setIfFeasible](#)₁₂₂ in class [ReleaseParameters](#)₁₁₆

Parameters:

`cost` - The proposed cost. to determine when any particular object exceeds `cost`. If null, the default value is a new instance of `RelativeTime(0,0)`.

`deadline` - The proposed deadline. If null, the default value is a new instance of `RelativeTime(Long.MAX_VALUE, 999999)`.

Returns: `false`. Aperiodic parameters never yield a feasible system.

(Subclasses of `AperiodicParameters`, such as [SporadicParameters](#)₁₃₆, need not return `false`.)

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of `cost` is less than zero, or the time value of `deadline` is less than or equal to zero, or the values are incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

`IllegalAssignmentError448` - Thrown if `cost` or `deadline` cannot be stored in this.

```
public void setInitialArrivalTimeQueueLength(
    int initial)
```

Sets the initial number of elements the arrival time queue can hold without lengthening the queue. The initial length of an arrival queue is set when the SO using the queue is constructed, after that time changes in the initial queue length are ignored.

Parameters:

`initial` - The initial length of the queue.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is less than zero.

Since: 1.0.1 Moved here from `SporadicParameters`.

6.10 SporadicParameters

Declaration

```
public class SporadicParameters extends AperiodicParameters129
```

All Implemented Interfaces: `java.lang.Cloneable`

Description

A notice to the scheduler that the associated schedulable object's run method will be released aperiodically but with a minimum time between releases.

When a reference to a `SporadicParameters` object is given as a parameter to a schedulable object's constructor or passed as an argument to one of the schedulable object's setter methods, the `SporadicParameters` object becomes the release parameters object bound to that schedulable object. Changes to the values in the `SporadicParameters` object affect that schedulable object. If bound to more than one

schedulable object then changes to the values in the `SporadicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

The implementation must use modified copy semantics for each `HighResolutionTime314` parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a `SporadicParameters` object caused by methods on that object cause the change to propagate to all schedulable objects using the parameter object. For instance, calling `setCost` on a `SporadicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the SOs that use the parameter object until a setter method on the `SporadicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

This class allows the application to specify one of four possible behaviors that indicate what to do if an arrival occurs that is closer in time to the previous arrival than the value given in this class as minimum interarrival time, what to do if, for any reason, the queue overflows, and the initial size of the queue.

Attribute	Value
minInterarrival time	No default. A value must be supplied
cost	<code>new RelativeTime(0,0)</code>
deadline	<code>new RelativeTime(mit)</code>
overrunHandler	None
missHandler	None
MIT violation policy	SAVE
Arrival queue overflow policy	SAVE
Initial arrival queue length	0

6.10.1 Fields

```
public static final java.lang.String mitViolationExcept
```

Represents the “EXCEPT” policy for dealing with minimum interarrival time violations. Under this policy, if an arrival time for any instance of [Schedulable₈₁](#) which has `this` as its instance of [ReleaseParameters₁₁₆](#) occurs at a time less than the minimum interarrival time defined here then the `fire()` method shall throw [MITViolationException₄₅₃](#). Any other associated semantics are governed by the schedulers for the schedulable objects using these sporadic parameters. If the arrival time is a result of a happening to which the instance of [AsyncEventHandler₃₉₃](#) is bound then the arrival time is ignored.

```
public static final java.lang.String mitViolationIgnore
```

Represents the “IGNORE” policy for dealing with minimum interarrival time violations. Under this policy, if an arrival time for any instance of [Schedulable₈₁](#) which has `this` as its instance of [ReleaseParameters₁₁₆](#) occurs at a time less than the minimum interarrival time defined here then the new arrival time is ignored. Any other associated semantics are governed by the schedulers for the schedulable objects using these sporadic parameters.

```
public static final java.lang.String mitViolationReplace
```

Represents the “REPLACE” policy for dealing with minimum interarrival time violations. Under this policy if an arrival time for any instance of [Schedulable₈₁](#) which has `this` as its instance of [ReleaseParameters₁₁₆](#) occurs at a time less than the minimum interarrival time defined here then the information for this arrival replaces a previous arrival. Any other associated semantics are governed by the schedulers for the schedulable objects using these sporadic parameters.

```
public static final java.lang.String mitViolationSave
```

Represents the “SAVE” policy for dealing with minimum interarrival time violations. Under this policy the arrival time for any instance of [Schedulable₈₁](#) which has `this` as its instance of [ReleaseParameters₁₁₆](#) is not compared to the specified minimum inter-

arrival time. Any other associated semantics are governed by the schedulers for the schedulable objects using these sporadic parameters.

6.10.2 Constructors

```
public SporadicParameters(  
    javax.realtime.RelativeTime332 minInterarrival)
```

Create a SporadicParameters object. This constructor is equivalent to: SporadicParameters(minInterarrival, null, null, null, null)

Parameters:

`minInterarrival` - The release times of the schedulable object will occur no closer than this interval. This time object is treated as if it were copied. Changes to `minInterarrival` will not effect the SporadicParameters object. There is no default value. If `minInterarrival` is null an illegal argument exception is thrown.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `minInterarrival` is null or its time value is not greater than zero.

`IllegalAssignmentError448` - Thrown if `minInterarrival` cannot be stored in this.

Since: 1.0.1

```
public SporadicParameters(  
    javax.realtime.RelativeTime332 minInterarrival,  
    javax.realtime.RelativeTime332 cost,  
    javax.realtime.RelativeTime332 deadline,  
    javax.realtime.AsyncEventHandler393 overrunHandler,  
    javax.realtime.AsyncEventHandler393 missHandler)
```

Create a SporadicParameters object.

Parameters:

`minInterarrival` - The release times of the schedulable object will occur no closer than this interval. This time object is treated as if it were copied. Changes to `minInterarrival` will not effect the SporadicParameters object. There is no default value. If `minInterarrival` is null an illegal argument exception is thrown.

cost - Processing time per release. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives per release. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds cost. If null, the default value is a new instance of `RelativeTime(0,0)`.

deadline - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. For a minimum implementation for purposes of feasibility analysis, the deadline is equal to the minimum interarrival interval. Other implementations may use this parameter to compute execution eligibility. If null, the default value is a new instance of `RelativeTime(mit)`.

overrunHandler - This handler is invoked if an invocation of the schedulable object exceeds cost. Not required for minimum implementation. If null no overrun handler will be used.

missHandler - 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 deadline miss handler will be used.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `minInterarrival` is null or its time value is not greater than zero, or the time value of `cost` is less than zero, or the time value of `deadline` is not greater than zero.

`IllegalAssignmentError448` - Thrown if `minInterarrival`, `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in this.

6.10.3 Methods

```
public javax.realtime.RelativeTime332
    getMinimumInterarrival()
```

Gets the minimum interarrival time.

Returns: The minimum interarrival time.

```
public java.lang.String getMitViolationBehavior()
```

Gets the arrival time queue behavior in the event of a minimum interarrival time violation.

Returns: The minimum interarrival time violation behavior as a string.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the new cost, and deadline as replacements for the matching attributes of this. If the resulting system is feasible, the method replaces the current scheduling characteristics, of this with the new scheduling characteristics.

Overrides: [setIfFeasible₁₃₅](#) in class [AperiodicParameters₁₂₉](#)

Parameters:

cost - The proposed cost. to determine when any particular object exceeds cost. If null, the default value is a new instance of `RelativeTime(0,0)`.

deadline - The proposed deadline. If null, the default value is a new instance of `RelativeTime(mit)`.

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the time value of cost is less than zero, or the time value of deadline is less than or equal to zero, or the values are incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

`IllegalAssignmentError448` - Thrown if cost or deadline cannot be stored in this.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 interarrival,
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the new interarrival, cost and deadline attributes as replacements for the matching attributes of this. If the resulting system is feasible the method replaces the current attributes with the new ones.

Changes to a `SporadicParameters` instance effect subsequent arrivals.

Parameters:

`interarrival` - The proposed interarrival time. There is no default value. If `minInterarrival` is null an illegal argument exception is thrown.

`cost` - The proposed cost. If null, the default value is a new instance of `RelativeTime(0, 0)`.

`deadline` - The proposed deadline. If null, the default value is a new instance of `RelativeTime(mit)`.

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `minInterarrival` is null or its time value is not greater than zero, or the time value of `cost` is less than zero, or the time value of `deadline` is not greater than zero.

`IllegalAssignmentError`₄₄₈ - Thrown if `interarrival`, `cost` or `deadline` cannot be stored in this.

```
public void setMinimumInterarrival(
    javax.realtime.RelativeTime332 minimum)
```

Set the minimum interarrival time.

Parameters:

`minimum` - The release times of the schedulable object will occur no closer than this interval.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `minimum` is null or its time value is not greater than zero.

`IllegalAssignmentError448` - Thrown if minimum cannot be stored in this.

```
public void setMitViolationBehavior(
    java.lang.String behavior)
```

Sets the behavior of the arrival time queue in the case where the new arrival time is closer to the previous arrival time than the minimum interarrival time given in this.

Values of `behavior` are compared using reference equality (`==`) not value equality (`equals()`).

Parameters:

`behavior` - A string representing the behavior.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `behavior` is not one of the final MIT violation behavior values defined in this class.

6.11 ProcessingGroupParameters

Declaration

```
public class ProcessingGroupParameters implements
    java.lang.Cloneable
```

All Implemented Interfaces: `java.lang.Cloneable`

Description

This is associated with one or more schedulable objects for which the system guarantees that the associated objects will not be given more time per period than indicated by `cost`. On implementations which do not support processing group parameters, this class may be used as a hint to the feasibility algorithm. The motivation for this class is to allow the execution demands of one or more aperiodic schedulable objects to be bound so that they can be included in feasibility analysis. However, periodic or sporadic schedulable objects can also be associated with a processing group.

For all schedulable objects with a reference to an instance of `ProcessingGroupParameters` `p` no more than `p.cost` will be allocated to the execution of these schedulable objects in each interval of time given by `p.period` after the time indicated by `p.start`. Logically a virtual server is associated with each

instance of `ProcessingGroupParameters`. This server has a start time, a period, a cost (budget) and a deadline. The server can only logically execute when (a) it has not consumed more execution time in its current release than the cost (budget) parameter, (b) one of its associated schedulable objects is executable and is the most eligible of the executable schedulable objects. If the server is logically executable, the associated schedulable object is executed. When the cost has been consumed, any `overrunHandler` is released, and the server is not eligible for logical execution until its next period is due. At this point, its allocated cost (budget) is replenished. If the server is logically executing when its deadline expires, any associated `missHandler` is released. The deadline and cost parameters of all the associated schedulable objects have the same impact as they would if the objects were not bound to a processing group.

Processing group parameters use `HighResolutionTime314` values for cost, deadline, period and start time. Since those times are expressed as a `HighResolutionTime314`, the values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity it measures depends on the clock associated with each time value.

When a reference to a `ProcessingGroupParameters` object is given as a parameter to a schedulable object's constructor or passed as an argument to one of the schedulable object's setter methods, the `ProcessingGroupParameters` object becomes the processing group parameters object bound to that schedulable object. Changes to the values in the `ProcessingGroupParameters` object affect that schedulable object. If bound to more than one schedulable object then changes to the values in the `ProcessingGroupParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

The implementation must use modified copy semantics for each `HighResolutionTime314` parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a `ProcessingGroupParameters` object caused by methods on that object are immediately visible to the scheduler. For instance, invoking `setPeriod()` on a `ProcessingGroupParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the scheduler's view of the processing group parameters object is updated. Invoking a method on the `RelativeTime` object that is the period for this object may change the period but it does not pass the change to the scheduler at that time. That new value for period must not change the behavior of the SOs that use the parameter object until a setter method on the `ProcessingGroupParameters` object is invoked, or the parameter object is used in `setProcessingGroupParameters()` or a constructor for an SO.

The implementation may use copy semantics for each `HighResolutionTime` parameter value. For instance the value returned by `getCost()` must be equal to the value passed in by `setCost`, but it need not be the same object.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Caution: The cost parameter time should be considered to be measured against the target platform.

Attribute	Default Value
start	new RelativeTime(0,0)
period	No default. A value must be supplied
cost	No default. A value must be supplied
deadline	new RelativeTime(period)
overrunHandler	None
missHandler	None

6.11.1 Constructors

```
public ProcessingGroupParameters(
    javax.realtime.HighResolutionTime314 start,
    javax.realtime.RelativeTime332 period,
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline,
    javax.realtime.AsyncEventHandler393 overrunHandler,
    javax.realtime.AsyncEventHandler393 missHandler)
```

Create a `ProcessingGroupParameters` object.

Parameters:

`start` - Time at which the first period begins. If a `RelativeTime`, this time is relative to the creation of this. If an `AbsoluteTime`, then the first release of the logical server is at the start time (or immediately if the absolute time is in the past). If null, the default value is a new instance of `RelativeTime(0,0)`.

`period` - The period is the interval between successive replenishment of the logical server's associated cost budget.

There is no default value. If `period` is null an exception is thrown.

`cost` - Processing time per period. The budget CPU time that the logical server can consume each period. If null, an exception is thrown.

`deadline` - The latest permissible completion time measured from the start of the current period. Changing the deadline might not take effect after the expiration of the current deadline. Specifying a deadline less than the period constrains execution of all the members of the group to the beginning of each period. If null, the default value is new instance of `RelativeTime(period)`.

`overrunHandler` - This handler is invoked if any schedulable object member of this processing group attempts to use processor time beyond the group's budget. If null, no application async event handler is fired on the overrun condition.

`missHandler` - This handler is invoked if the logical server is still executing after the deadline has passed. If null, no application async event handler is fired on the deadline miss condition.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `period` is null or its time value is not greater than zero, if `cost` is null, or if the time value of `cost` is less than zero, if `start` is an instance of `RelativeTime` and its value is negative, or if the time value of `deadline` is not greater than zero and less than or equal to the `period`. If the implementation does not support processing group deadline less than period, `deadline` less than period will cause `IllegalArgumentException` to be thrown.

`IllegalAssignmentError448` - Thrown if `start`, `period`, `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in this.

6.11.2 Methods

```
public java.lang.Object clone()
```

Return a clone of `this`. This method should behave effectively as if it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Overrides: `clone` in class `Object`

Since: 1.0.1

```
public javax.realtime.RelativeTime332 getCost()
```

Gets the value of `cost`.

Returns: a reference to the value of `cost`.

```
public javax.realtime.AsyncEventHandler393
    getCostOverrunHandler()
```

Gets the cost overrun handler.

Returns: A reference to an instance of `AsyncEventHandler393` that is cost overrun handler of `this`.

```
public javax.realtime.RelativeTime332 getDeadline()
```

Gets the value of `deadline`.

Returns: A reference to an instance of `RelativeTime332` that is the deadline of `this`.

```
public javax.realtime.AsyncEventHandler393
    getDeadlineMissHandler()
```

Gets the deadline miss handler.

Returns: A reference to an instance of `AsyncEventHandler393` that is deadline miss handler of `this`.

```
public javax.realtime.RelativeTime332 getPeriod()
```

Gets the value of `period`.

Returns: A reference to an instance of `RelativeTime332` that represents the value of `period`.

```
public javax.realtime.HighResolutionTime314 getStart()
```

Gets the value of `start`. This is the value that was specified in the constructor or by `setStart()`, not the actual absolute time the corresponding to the start of the processing group.

Returns: A reference to an instance of [HighResolutionTime](#)₃₁₄ that represents the value of `start`.

```
public void setCost(javax.realtime.RelativeTime332 cost)
```

Sets the value of `cost`.

Parameters:

`cost` - The new value for `cost`. If null, an exception is thrown.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if `cost` is null or its time value is less than zero.

[IllegalAssignmentError](#)₄₄₈ - Thrown if `cost` cannot be stored in this.

```
public void setCostOverrunHandler(  
    javax.realtime.AsyncEventHandler393 handler)
```

Sets the cost overrun handler.

Parameters:

`handler` - This handler is invoked if the `run()` method of and of the the schedulable objects attempt to execute for more than `cost` time units in any period. If null, no handler is attached, and any previous handler is removed.

Throws:

[IllegalAssignmentError](#)₄₄₈ - Thrown if `handler` cannot be stored in this.

```
public void setDeadline(  
    javax.realtime.RelativeTime332 deadline)
```

Sets the value of `deadline`.

Parameters:

`deadline` - The new value for `deadline`. If null, the default value is new instance of `RelativeTime(period)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `deadline` has a value less than zero or greater than the period. Unless the implementation supports deadline less than period in processing groups, `IllegalArgumentException` is also thrown if `deadline` is less than the period.

`IllegalAssignmentError448` - Thrown if `deadline` cannot be stored in this.

```
public void setDeadlineMissHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Sets the deadline miss handler.

Parameters:

`handler` - This handler is invoked if the `run()` method of any of the schedulable objects still expect to execute after the deadline has passed. If null, no handler is attached, and any previous handler is removed.

Throws:

`IllegalAssignmentError448` - Thrown if `handler` cannot be stored in this.

```
public boolean setIfFeasible(
    javax.realtime.RelativeTime332 period,
    javax.realtime.RelativeTime332 cost,
    javax.realtime.RelativeTime332 deadline)
```

This method first performs a feasibility analysis using the period, cost and deadline attributes as replacements for the matching attributes of `this`. If the resulting system is feasible the method replaces the current attributes of `this` with the new attributes.

Parameters:

`period` - The proposed period. There is no default value. If `period` is null an exception is thrown.

`cost` - The proposed cost. If null, an exception is thrown.

`deadline` - The proposed deadline. If null, the default value is new instance of `RelativeTime(period)`.

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown 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.

`IllegalAssignmentError448` - Thrown if `period`, `cost`, or `deadline` cannot be stored in this.

public void setPeriod(`javax.realtime.RelativeTime332` period)

Sets the value of `period`.

Parameters:

`period` - The new value for `period`. There is no default value. If `period` is null an exception is thrown.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `period` is null, or its time value is not greater than zero. If the implementation does not support processing group deadline less than `period`, and `period` is not equal to the current value of the processing group's deadline, the deadline is set to a clone of `period` created in the same memory area as `period`.

`IllegalAssignmentError448` - Thrown if `period` cannot be stored in this.

**public void setStart(
`javax.realtime.HighResolutionTime314` start)**

Sets the value of `start`. If the processing group is already started this method alters the value of this object's start time property, but has no other effect.

Parameters:

`start` - The new value for `start`. If null, the default value is a new instance of `RelativeTime(0,0)`.

Throws:

`IllegalAssignmentError448` - Thrown if `start` cannot be stored in this.

`java.lang.IllegalArgumentException` - Thrown if `start` is a relative time value and less than zero.

Memory Management

This section defines classes directly related to memory and memory management. These classes:

- Allow the definition of regions of memory outside of the traditional Java heap.
- Allow the definition of regions of scoped memory, that is, memory regions with a limited lifetime.
- Allow the definition of regions of memory containing objects whose lifetime matches that of the application.
- Allow the definition of regions of memory mapped to specific physical addresses.
- Allow the specification of maximum memory area consumption and maximum allocation rates for individual schedulable objects.
- Allow the programmer to query information characterizing the behavior of the garbage collection algorithm, and to some limited ability, alter the behavior of that algorithm.

Definitions

Schedulable objects that use the `enter` method of `MemoryArea` behave effectively as if they kept the memory areas they enter in a *scope stack* which `enter` pushes and `pops`.

This chapter defines memory area classes. Two memory areas may be associated with each `MemoryArea` instance, the memory area containing the instance and the *backing memory* that contains memory managed by the `MemoryArea` instance.

Some memory area classes implement *portals*. These are a tool that associates a reference value with a memory area. It is normally used to give code that has a reference to the memory area a way to go from that to a reference to an object stored in that memory area.

For purposes of scoped memory reference counting, the following are treated as *execution contexts*:

- `RealtimeThread` objects that have been started and have not terminated,
- `AsyncEventHandler` objects that are currently in a released state,
- `AsyncEvent` objects that are bound to happenings,
- `Timer` objects that have been started and have not been destroyed,
- other schedulable objects that control an execution engine

The initial memory area for a schedulable object is *non-default* if it is not the memory area where the schedulable object was created.

An `AsyncEventHandler` is *fireable* whenever there is an agent that can release it. This includes cases when the `AsyncEventHandler` is:

- A miss handler, or overrun handler for a real-time thread that has been started but not yet terminated
- A miss handler or overrun handler for an `AsyncEventHandler` that is itself fireable;
- A handler associated with an `AsyncEvent` bound to a happening;
- A handler associated with a `Timer` that has been started but not yet destroyed;
- A handler associated with an `AsyncEvent` that can be programmatically fired

Semantics and Requirements

The following list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

Allocation time

1. Some `MemoryArea` classes are required to have linear (in object size) allocation time. The linear time attribute requires that, ignoring performance variations due

to hardware caches or similar optimizations and ignoring execution time of any static initializers, the execution time of `new` must be bounded by a polynomial, $f(n)$, where n is the size of the object and for all $n > 0$, $f(n) \leq Cn$ for some constant C .

2. Execution time of object constructors, and time spent in class loading and static initialization are not governed by bounds on object allocation in this specification, but setting default initial values for fields in the instance (as specified in *The Java Virtual Machine Specification*, Second Edition, section 2.5.1, “Each class variable, instance variable, and array component is initialized with a default value when it is created.”) is considered part of object allocation and included in the time bound.

The allocation context

3. A memory area is represented by an instance of a subclass of the `MemoryArea` class. When a memory area, m , is entered by calling `m.enter` (or another method from the family of enter-like methods in `MemoryArea` or `ScopedMemory`) m becomes the *allocation context* of the current schedulable object. When control returns from the `enter` method, the allocation context is restored to the value it had immediately before `enter` was called.
4. When a memory area, m , is entered by calling m 's `executeInArea` method, m becomes the current allocation context of the current schedulable object. When control returns from the `executeInArea` method, the allocation context is restored to the value it had before `executeInArea` was called.
5. The initial allocation context for a schedulable object when it is first released, is the memory area that was designated the *initial memory area* when the schedulable object was constructed. This initial allocation context becomes the current allocation context for that schedulable object when the schedulable object first becomes eligible for execution. For async event handlers, the initial allocation context is the same on each release; for real-time threads, in releases subsequent to the first, the allocation context is the same as it was when the real-time thread became *blocked-for-release-event*.
6. All object allocation through the `new` keyword will use the current allocation context, but note that allocation can be performed in a specific memory area using the `newInstance` and `newArray` methods.
7. Schedulable objects behave as if they stored their memory area context in a structure called the *scope stack*. This structure is manipulated by creation of schedulable objects, and the following methods from the `MemoryArea` and `ScopedMemory` classes: all the `enter` and `joinAndEnter` methods, `executeInArea`, and both `newInstance` methods. See the semantics in *Maintaining the Scope Stack* for

details.

8. The scope stack is accessible through a set of static methods on `RealtimeThread`. These methods allow outer allocation contexts to be accessed by their index number. Memory areas on a scope stack may be referred to as *inner* or *outer* relative to other entries in that scope stack. An “outer scope” is further from the current allocation context on the current scope stack and has a lower index.
9. The `executeInArea`, `newInstance` and `newArray` methods, when invoked on an instance of `ScopedMemory` require that instance to be an outer allocation context on the current schedulable object’s current scope stack.
10. An instance of `ScopedMemory` is said to be *in use* if it has a non-zero reference count as defined by semantic (17) below.

The Parent Scope

11. Instances of `ScopedMemory` have special semantics including definition of *parent*. If a `ScopedMemory` object is neither in use nor the initial memory area for a schedulable object, it has no *parent* scope.
 - When a `ScopedMemory` object becomes in use, its parent is the nearest `ScopedMemory` object outside it on the current scope stack. If there is no outside `ScopedMemory` object in the current scope stack, the parent is the *primordial scope* which is not actually a memory area, but only a marker that constrains the parentage of `ScopedMemory` objects.
 - At construction of a schedulable object, if the initial memory area has no parent, the initial memory area is assigned the parent it will have when the schedulable object is in execution. This rule determines the initial memory area’s parent until the schedulable object is de-allocated from its memory area, or, if the schedulable object is a `RealtimeThread`, it completes execution
12. Instances of `ScopedMemory` must satisfy the *single parent rule* which requires that each scoped memory has a unique parent as defined in semantic (11.)

Memory areas and schedulable objects

13. Pushing a scoped memory onto a scope stack is always subject to the single parent rule.
14. Each schedulable object has an initial memory area which is that object’s initial allocation context. The default initial memory area is the current allocation context in effect during execution of the schedulable object’s constructor, but schedulable objects may supply constructors that override the default.
15. A Java thread cannot have a scope stack; consequently it can only be created and

execute within heap or immortal memory. An attempt to create a Java thread in a scoped memory area throws `IllegalAssignmentError`.

16. A Java thread may use `executeInArea`, and the `newInstance` and `newArray` methods from the `ImmortalMemory` and `HeapMemory` classes. These methods allow it to execute with an immortal current allocation context, but semantic (15) applies even during execution of these methods.

Scoped memory reference counting

17. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count which is greater than zero if and only if either:
 - the scoped memory area is the current allocation context or an outer allocation context for one or more *execution contexts*; or else
 - the scoped memory area is the *non-default initial memory area* for a *fireable AsyncEventHandler*.
18. When the reference count for an instance of the class `ScopedMemory` is ready to be decremented from one to zero, all unfinalized objects within that area are considered ready for finalization. If after the finalizers for all such unfinalized objects in the scoped memory area run to completion, the reference count for the memory area is still ready to be decremented to zero, any newly created unfinalized objects are considered ready for finalization and the process is repeated until no new objects are created or the scoped memory's reference count is no longer ready to be decremented from one to zero. When the scope contains no unfinalized objects and its reference count is ready to be decremented from one to zero, then the reference count is decremented to zero and the memory scope is emptied of all objects. The RTSJ implementation must complete finalization of objects in the scope and, if the reference count is zero after finalizers run, deletion of the objects in the scope before that memory scope can again become the current allocation context for any schedulable object. (This is a special case of the finalization implementation specified in *The Java Language Specification*, second edition, section 12.6.1)
19. Finalization may start when all unfinalized objects in the scope are ready for finalization. Finalizers are executed with the current allocation context set to the finalizing scope and are executed by the schedulable object in control of the scope when its reference count is ready to be decremented from one to zero. If finalizers are executed because a real-time thread terminates or an `AsyncEventHandler` becomes non-fireable, that real-time thread or `AsyncEventHandler` is considered in control of the scope and must execute the finalizers.
20. From the time objects in a scope are deleted until the portal on the scope is successfully set to a non-null value with `setPortal`, the value returned by `get-`

Portals on that scoped memory object must be null.

Immortal memory

21. Objects created in any immortal memory area are unexceptionally referenceable from all Java threads, and all schedulable objects, and the allocation and use of objects in immortal memory is never subject to garbage collection delays.
22. An implementation may execute finalizers for immortal objects when it determines that the application has terminated. Finalizers will be executed by a thread or schedulable object whose current allocation context is not scoped memory. Regardless of any call to `runFinalizersOnExit`, the system need not execute finalizers for immortal objects that remain unfinalized when the JVM begins termination.
23. Class objects, the associated static memory, and interned Strings behave effectively as if they were allocated in immortal memory with respect to reference rules, assignment rules, and preemption delays by no-heap schedulable objects. Static initializers are executed effectively as if the current thread performed `ImmutableMemory.instance().executeInArea(r)` where `r` is a `Runnable` that executes the `<clinit>` method of the class being initialized.

Maintaining referential integrity

24. Assignment rules placed on reference assignments prevent the creation of dangling references, and thus maintain the referential integrity of the Java runtime. The restrictions are listed in the following table:

Stored In	Reference to Heap	Reference to Immortal	Reference to Scoped	null
Heap	Permit	Permit	Forbid	Permit
Immortal	Permit	Permit	Forbid	Permit
Scoped	Permit	Permit	Permit, if the reference is from the same scope, or an outer scope	Permit
Local Variable	Permit	Permit	Permit	Permit

For this table, `ImmutableMemory` and `ImmutablePhysicalMemory` are equivalent, and all sub-classes of `ScopedMemory` are equivalent.

25. An implementation must ensure that the above checks are performed on every assignment statement before the statement is executed. (This includes the possibility of static analysis of the application logic). Checks for operations on local variables are not required because a potentially invalid reference would be captured by the other checks before it reached a local variable.

Object initialization

26. Static initializers run with the immortal memory area as their allocation context.
27. The current allocation context in a constructor for an object is the memory area in which the object is allocated. For `new`, this is the current allocation context when `new` was called. For members of the `m.newInstance` family, the current allocation context is memory area `m`.

Maintaining the Scope Stack

This section describes maintenance of a data structure that is called the *scope stack*. Implementations are not required to use a stack or implement the algorithms given here. It is only required that an implementation behave with respect to the ordering and accessibility of memory scopes effectively as if it implemented these algorithms.

The scope stack is implicitly visible through the assignment rules, and the stack is explicitly visible through the static `getOuterMemoryArea(int index)` method on `RealtimeThread`.

Four operations effect the scope stack: the `enter` methods in `MemoryArea` and `ScopedMemory`, construction of a new schedulable object, the `executeInArea` method in `MemoryArea`, and the `newInstance` methods in `MemoryArea`.

28. The memory area at the top of a schedulable object's scope stack is the schedulable object's current allocation context.
29. When a schedulable object, t , creates a schedulable object, n_t , in a `ScopedMemory` object's allocation area, n_t acquires a copy of the scope stack associated with t at the time n_t is constructed including all entries from up to and including the memory area containing n_t . If n_t is created in heap, immortal, or immortal physical memory, n_t is created with a scope stack containing only heap, immortal, or immortal physical memory respectively. If n_t has a non-default initial memory area, `ima`, then `ima` is pushed on n_t 's newly-created scope stack.
30. When a memory area, `ma`, is entered by calling a `ma.enter` method, `ma` is pushed on the scope stack and becomes the *allocation context* of the current schedulable object. When control returns from the `enter` method, the allocation context is popped from the scope stack

31. When a memory area, *m*, is entered by calling *m*'s `executeInArea` method or one of the `m.newInstance` methods the scope stack before the method call is preserved and replaced with a scope stack constructed as follows:
- If *ma* is a scoped memory area the new scope stack is a copy of the schedulable object's previous scope stack up to and including *ma*.
 - If *ma* is not a scoped memory area the new scope stack includes only *ma*.

When control returns from the `executeInArea` method, the scope stack is restored to the value it had before `ma.executeInArea` or `ma.newInstance` was called.

Notes:

- For the purposes of these algorithms, stacks grow *up*.
- The representative algorithms ignore important issues like freeing objects in scopes.
- In every case, objects in a scoped memory area are eligible to be freed when the reference count for the area is zero after finalizers for that scope are run.
- Informally, any objects in a scoped memory area *must* be freed and their finalizers run before the reference count for the memory area is incremented from zero to one.

enter

```
For ma.enter(logic):
push ma on the scope stack belonging to the current
    schedulable object -- which may throw
    ScopedCycleException
execute logic.run method
pop ma from the scope stack
```

executeInArea or newInstance

For `ma.executeInArea(logic)`, `ma.newInstance()`, or `ma.newArray()`:

```
if ma is an instance of heap immortal or
    ImmortalPhysicalMemory
    start a new scope stack containing only ma
    make the new scope stack the scope stack for
        the current schedulable object
else ma is scoped
    if ma is in the scope stack for the
        current schedulable object
        start a new scope stack containing ma and all
            scopes below ma on the scope stack.
        make the new scope stack the scope stack for
            the current schedulable object
    else
        throw InaccessibleAreaException
execute logic.run or construct the object
restore the previous scope stack for the
    current schedulable object
discard the new scope stack
```

Construct a Schedulable Object

For construction of a schedulable object in memory area *cma* with initial memory area of *ima*:

```

if cma is heap, immortal or ImmortalPhysicalMemory
    create a new scope stack containing cma
else
    start a new scope stack containing the
        entire current scope stack
if ima != cma
    push ima on the new scope stack --
        which may throw ScopedCycleException

```

The above pseudocode illustrates a straightforward implementation of this specification's semantics, but any implementation that behaves effectively like this one with respect to reference count values of zero and one is permissible. An implementation may be eager or lazy in maintenance of its reference count provided that it correctly implements the semantics for reference counts of zero and one.

The Single Parent Rule

Every push of a scoped memory type on a scope stack requires reference to the single parent rule, this enforces the invariant that every scoped memory area has no more than one parent.

The parent of a scoped memory area is identified by the following rules (for a stack that grows up):

- If the memory area is not currently on any scope stack, it has no parent
- If the memory area is the outermost (lowest) scoped memory area on any scope stack, its parent is the *primordial scope*.
- For all other scoped memory areas, the parent is the first scoped memory area outside it on the scope stack.

Except for the primordial scope, which represents heap, immortal and immortal physical memory, only scoped memory areas are visible to the single parent rule.

The operational effect of the single parent rule is that when a scoped memory area has a parent, the only legal change to that value is to "no parent." Thus an ordering imposed by the first assignments of parents of a series of nested scoped memory areas is the only nesting order allowed until control leaves the scopes; then a new nesting order is possible. Thus a schedulable object attempting to enter a scope can only do so by entering in the established nesting order.

Scope Tree Maintenance

The single parent rule is enforced effectively as if there were a tree with the primordial scope (representing heap, immortal, and immortal physical memory) at its

root, and other nodes corresponding to every scoped memory area that is currently on any schedulable object's scope stack.

Each scoped memory has a reference to its parent memory area, `ma.parent`. The parent reference may indicate a specific scoped memory area, no parent, or the primordial parent.

If a scoped memory area is the non-default initial memory area of an async event handler, or the non-default initial memory area of a real-time thread that has not terminated, it is referred to as *pinned*.

On Scope Stack Push of `ma`

The following procedure could be used to maintain the scope tree and ensure that push operations on a schedulable object's scope stack do not violate the single parent rule.

```
precondition: ma.parent is set to the correct parent
              (either a scoped memory area or the primordial scope)
              or to noParent
t.scopeStack is the scope stack of
the current schedulable object
if ma is scoped
  parent = findFirstScope(t.scopeStack)
  if ma.parent == noParent
    ma.parent = parent
  else if ma.parent != parent
    throw ScopedCycleException
  else
    t.scopeStack.push(ma)
```

`findFirstScope` is a convenience function that looks down the scope stack for the next entry that is a reference to an instance of `ScopedMemoryArea`.

```
findFirstScope(scopeStack) {
  for s = top of scope stack to
    bottom of scope stack
    if s is an instance of ScopedMemory
      return s
  return primordial scope
}
```

On Scope Stack Pop of `ma`

```
ma = t.scopeStack.pop()
if ma is scoped
  if !(ma.in_use || ma.pinned)
    ma.parent = noParent
```

The Rationale

Languages that employ automatic reclamation of blocks of memory allocated in what is traditionally called the heap by program logic also typically use an algorithm called a garbage collector. Garbage collection algorithms and implementations vary in the amount of non-determinacy they add to the execution of program logic. Rather than require a garbage collector, and require it to meet real-time constraints that would necessarily be a compromise, this specification constructs alternative systems for

“safe” management of memory. The scoped and immortal memory areas allow program logic to allocate objects in a Java-like style, ignore the reclamation of those objects, and not incur the latency of the implemented garbage collection algorithm.

The term *scope stack* might mislead a reader to infer that it contains only scoped memory areas. This is incorrect. Although the scope stack may contain scoped memory references, it may also contain heap and immortal memory areas. Also, although the scope stack’s behavior is specified as a stack, an implementation is free to use any data structure that preserves the stack semantics.

This specification does not specifically address the lifetime of objects allocated in immortal memory areas. If they were reclaimed while they were still referenced, the referential integrity of the JVM would be compromised which is not permissible. Recovering immortal objects only at the termination of the application, or never recovering them under any circumstances is consistent with this specification.

If a scoped memory area is used by both heap and non-heap SOs, there could be cases where a finalizer executed in non-heap context could attempt to use a heap reference left by a heap-using SO. The code in the finalizer would throw a memory access error. If that exception is not caught in the finalizer, it will be handled by the implementation so finalization will continue undisturbed, but the problem in finalizer that caused the illegal memory access could be hard to locate. So, catch clauses in finalizers for objects allocated in scoped memory are even more useful than they are for normal finalizers.

7.1 MemoryArea

Declaration

```
public abstract class MemoryArea
```

Direct Known Subclasses: [HeapMemory₁₆₈](#), [ImmortalMemory₁₆₈](#),
[ImmortalPhysicalMemory₂₁₂](#), [ScopedMemory₁₇₂](#)

Description

MemoryArea is the abstract base class of all classes dealing with the representations of allocatable memory areas, including the immortal memory area, physical memory and scoped memory areas. This is an abstract class, but no method in this class is abstract. An application should not subclass MemoryArea without complete knowledge of its implementation details.

7.1.1 Constructors

protected MemoryArea(long size)

Create an instance of MemoryArea.

Parameters:

size - The size of MemoryArea to allocate, in bytes.

Throws:

java.lang.IllegalArgumentException - Thrown if size is less than zero.

java.lang.OutOfMemoryError - Thrown if there is insufficient memory for the MemoryArea object or for the backing memory.

**protected MemoryArea(
 long size,
 java.lang.Runnable logic)**

Create an instance of MemoryArea.

Parameters:

size - The size of MemoryArea to allocate, in bytes.

logic - The run() method of this object will be called whenever `enter()`₁₆₃ is called. If logic is null, this constructor is equivalent to MemoryArea(long size).

Throws:

java.lang.IllegalArgumentException - Thrown if the size parameter is less than zero.

java.lang.OutOfMemoryError - Thrown if there is insufficient memory for the MemoryArea object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

protected MemoryArea(javax.realtime.SizeEstimator₁₇₀ size)

Create an instance of MemoryArea.

Parameters:

size - A `SizeEstimator`₁₇₀ object which indicates the amount of memory required by this MemoryArea.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the size parameter is null, or `size.getEstimate()` is negative.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `MemoryArea` object or for the backing memory.

protected MemoryArea(

`javax.realtime.SizeEstimator`₁₇₀ size,
`java.lang.Runnable` logic)

Create an instance of `MemoryArea`.

Parameters:

size - A `SizeEstimator` object which indicates the amount of memory required by this `MemoryArea`.

logic - The `run()` method of this object will be called whenever `enter()`₁₆₃ is called. If logic is null, this constructor is equivalent to `MemoryArea(SizeEstimator size)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if size is null or `size.getEstimate()` is negative.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `MemoryArea` object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

7.1.2 Methods

public void enter()

Associate this memory area with the current schedulable object for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea(Runnable)`₁₆₅) or the `enter` method exits.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and no non-null value for `logic` was supplied when the memory area was constructed.

`ThrowBoundaryError`₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`₄₄₈, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`₄₅₆ is allocated in the current allocation context and contains information about the exception it replaces.

`MemoryAccessError`₄₅₀ - Thrown if caller is a no-heap schedulable object and this memory area's logic value is allocated in heap memory.

public void enter(`java.lang.Runnable logic`)

Associate this memory area with the current schedulable object for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea(Runnable)`₁₆₅) or the `enter` method exits.

Parameters:

`logic` - The `Runnable` object whose `run()` method should be invoked.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and `logic` is null.

`ThrowBoundaryError`₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`₄₄₈, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`₄₅₆ is allocated in the current allocation context and contains information about the exception it replaces.

```
public void executeInArea(java.lang.Runnable logic)
```

Execute the run method from the `logic` parameter using this memory area as the current allocation context. The effect of `executeInArea` on the scope stack is specified in the subclasses of `MemoryArea`.

Parameters:

`logic` - The runnable object whose `run()` method should be executed.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `logic` is null.

```
public static javax.realtime.MemoryArea161 getMemoryArea(  
    java.lang.Object object)
```

Gets the `MemoryArea` in which the given object is located.

Returns: The instance of `MemoryArea` from which object was allocated.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the value of object is null.

```
public long memoryConsumed()
```

For memory areas where memory is freed under program control this returns an exact count, in bytes, of the memory currently used by the system for the allocated objects. For memory areas (such as heap) where the definition of “used” is imprecise, this returns the best value it can generate in constant time.

Returns: The amount of memory consumed in bytes.

```
public long memoryRemaining()
```

An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

Returns: The amount of remaining memory in bytes.

```
public java.lang.Object newArray(  
    java.lang.Class type,  
    int number)
```

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Parameters:

type - The class of the elements of the new array. To create an array of a primitive type use a type such as `Integer.TYPE` (which would call for an array of the primitive `int` type.)

number - The number of elements in the new array.

Returns: A new array of class `type`, of `number` elements.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `number` is less than zero, `type` is null, or `type` is `java.lang.Void.TYPE`.

`java.lang.OutOfMemoryError` - Thrown if space in the memory area is exhausted.

```
public java.lang.Object newInstance(java.lang.Class type)
    throws InstantiationException,
           IllegalAccessException
```

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters:

type - The class of which to create a new instance.

Returns: A new instance of class `type`.

Throws:

`java.lang.IllegalAccessException` - The class or initializer is inaccessible.

`java.lang.IllegalArgumentException` - Thrown if `type` is null.

`java.lang.InstantiationException` - Thrown if the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array, or an exception was thrown by the constructor.

`java.lang.OutOfMemoryError` - Thrown if space in the memory area is exhausted.

`java.lang.ExceptionInInitializerError` - Thrown if an unexpected exception has occurred in a static initializer

`java.lang.SecurityException` - Thrown if the caller does not have permission to create a new instance.

```
public java.lang.Object newInstance(
    java.lang.reflect.Constructor c,
    java.lang.Object[] args)
    throws IllegalAccessException,
    InstantiationException, InvocationTargetException
```

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Parameters:

c - The constructor for the new instance.

args - An array of arguments to pass to the constructor.

Returns: A new instance of the object constructed by c.

Throws:

java.lang.ExceptionInInitializerError - Thrown if an unexpected exception has occurred in a static initializer

java.lang.IllegalAccessException - Thrown if the class or initializer is inaccessible under Java access control.

java.lang.IllegalArgumentException - Thrown if c is null, or the args array does not contain the number of arguments required by c. A null value of args is treated like an array of length 0.

java.lang.InstantiationException - Thrown if the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array.

java.lang.reflect.InvocationTargetException - Thrown if the underlying constructor throws an exception.

java.lang.OutOfMemoryError - Thrown if space in the memory area is exhausted.

```
public long size()
```

Query the size of the memory area. The returned value is the current size. Current size may be larger than initial size for those areas that are allowed to grow.

Returns: The size of the memory area in bytes.

7.2 HeapMemory

Declaration

```
public final class HeapMemory extends MemoryArea161
```

Description

The `HeapMemory` class is a singleton object that allows logic with a non-heap allocation context to allocate objects in the Java heap.

7.2.1 Methods

```
public void executeInArea(java.lang.Runnable logic)
```

Execute the `run` method from the `logic` parameter using heap as the current allocation context. For a schedulable object, this saves the current scope stack and replaces it with one consisting only of the `HeapMemory` instance; restoring the original scope stack upon completion.

Overrides: `executeInArea165` in class [MemoryArea₁₆₁](#)

Parameters:

`logic` - The runnable object whose `run()` method should be executed.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `logic` is null.

```
public static javax.realtime.HeapMemory168 instance()
```

Returns a reference to the singleton instance of [HeapMemory₁₆₈](#) representing the Java heap. The singleton instance of this class shall be allocated in the [ImmortalMemory₁₆₈](#) area.

Returns: The singleton [HeapMemory₁₆₈](#) object.

7.3 ImmortalMemory

Declaration

```
public final class ImmortalMemory extends MemoryArea161
```

Description

`ImmortalMemory` is a memory resource that is unexceptionally available to all schedulable objects and Java threads for use and allocation.

An immortal object may not contain references to any form of scoped memory, e.g., `VTMemory193`, `LTMemory187`, `VTPhysicalMemory230`, or `LTPhysicalMemory221`.

Objects in immortal have the same states with respect to finalization as objects in the standard Java heap, but there is no assurance that immortal objects will be finalized even when the JVM is terminated.

Methods from `ImmortalMemory` should be overridden only by methods that use `super`.

7.3.1 Methods

```
public void executeInArea(java.lang.Runnable logic)
```

Execute the run method from the `logic` parameter using this memory area as the current allocation context. For a schedulable object, this saves the current scope stack and replaces it with one consisting only of the `ImmortalMemory` instance; restoring the original scope stack upon completion.

Overrides: `executeInArea165` in class `MemoryArea161`

Parameters:

`logic` - The runnable object whose `run()` method should be executed.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `logic` is null.

```
public static javax.realtime.ImmortalMemory168 instance()
```

Returns a pointer to the singleton `ImmortalMemory168` object.

Returns: The singleton `ImmortalMemory168` object.

7.4 SizeEstimator

Declaration

```
public final class SizeEstimator
```

Description

This class maintains an estimate of the amount of memory required to store a set of objects.

`SizeEstimator` is a floor on the amount of memory that should be allocated. Many objects allocate other objects when they are constructed. `SizeEstimator` only estimates the memory requirement of the object itself, it does not include memory required for any objects allocated at construction time. If the instance itself is allocated in several parts (if for instance the object and its monitor are separate), the size estimate shall include the sum of the sizes of all the parts that are allocated from the same memory area as the instance.

Alignment considerations, and possibly other order-dependent issues 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.

See Also: [MemoryArea.MemoryArea\(SizeEstimator\)₁₆₂](#), [LTMemory.LTMemory\(SizeEstimator, SizeEstimator\)₁₉₁](#), [VTMemory.VTMemory\(SizeEstimator, SizeEstimator\)₁₉₆](#)

7.4.1 Constructors

```
public SizeEstimator()
```

7.4.2 Methods

```
public long getEstimate()
```

Gets an estimate of the number of bytes needed to store all the objects reserved.

Returns: The estimated size in bytes.

```
public void reserve(
    java.lang.Class c,
    int number)
```

Take into account additional `number` instances of Class `c` when estimating the size of the [MemoryArea₁₆₁](#) .

Parameters:

`c` - The class to take into account.

`number` - The number of instances of `c` to estimate.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `c` is null.

```
public void reserve(javax.realtime.SizeEstimator170 size)
```

Take into account an additional instance of `SizeEstimator` `size` when estimating the size of the [MemoryArea₁₆₁](#) .

Parameters:

`size` - The given instance of `SizeEstimator`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null.

```
public void reserve(
    javax.realtime.SizeEstimator170 estimator,
    int number)
```

Take into account additional `number` instances of `SizeEstimator` `size` when estimating the size of the [MemoryArea₁₆₁](#) .

Parameters:

`estimator` - The given instance of [SizeEstimator₁₇₀](#) .

`number` - The number of times to reserve the size denoted by `estimator`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `estimator` is null.

```
public void reserveArray(int length)
```

Take into account an additional instance of an array of `length` reference values when estimating the size of the [MemoryArea₁₆₁](#) .

Parameters:

`length` - The number of entries in the array.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `length` is negative.

Since: 1.0.1

```
public void reserveArray(
    int length,
    java.lang.Class type)
```

Take into account an additional instance of an array of `length` primitive values when estimating the size of the [MemoryArea₁₆₁](#).

Class values for the primitive types are available from the corresponding class types; e.g., `Byte.TYPE`, `Integer.TYPE`, and `Short.TYPE`.

Parameters:

`length` - The number of entries in the array.

`type` - The class representing a primitive type. The reservation will leave room for an array of `length` of the primitive type corresponding to `type`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `length` is negative, or `type` does not represent a primitive type.

Since: 1.0.1

7.5 ScopedMemory

Declaration

```
public abstract class ScopedMemory extends MemoryArea161
```

Direct Known Subclasses: [LTMemory₁₈₇](#), [LTPhysicalMemory₂₂₁](#), [VTMemory₁₉₃](#), [VTPhysicalMemory₂₃₀](#)

Description

`ScopedMemory` is the abstract base class of all classes dealing with representations of memory spaces which have a limited lifetime. In general, objects allocated in scoped memory are freed when (and only when) no schedulable object has access to the objects in the scoped memory.

A `ScopedMemory` area is a connection to a particular region of memory and reflects the current status of that memory. The object does not necessarily contain direct references to the region of memory. That is implementation dependent.

When a `ScopedMemory` area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents (it's backing store) is allocated from memory that is not otherwise directly visible to Java code; e.g., it might be allocated with the C `malloc` function. This backing store behaves effectively as if it were allocated when the associated scoped memory object is constructed and freed at that scoped memory object's finalization.

The `enter()`¹⁷⁵ method of `ScopedMemory` is one mechanism used to make a memory area the current allocation context. The other mechanism for activating a memory area is making it the initial memory area for a real-time thread or async event handler. Entry into the scope is accomplished, for example, by calling the method:

```
public void enter(Runnable logic)
```

where `logic` is an instance of `java.lang.Runnable` whose `run()` method represents the entry point of the code that will run in the new scope. Exit from the scope occurs between the time the `runnable.run()` method completes and the time control returns from the `enter` method. By default, allocations of objects within `runnable.run()` are taken from the backing store of the `ScopedMemory`.

`ScopedMemory` is an abstract class, but all specified methods include implementations. The responsibilities of `MemoryArea`, `ScopedMemory` and the classes that extend `ScopedMemory` are not specified. Application code should not extend `ScopedMemory` without detailed knowledge of its implementation.

7.5.1 Constructors

```
public ScopedMemory(long size)
```

Create a new `ScopedMemory` area with the given parameters.

Parameters:

`size` - The size of the new `ScopedMemory` area in bytes.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ScopedMemory` object or for the backing memory.

```
public ScopedMemory(
    long size,
    java.lang.Runnable logic)
```

Create a new ScopedMemory area with the given parameters.

Parameters:

`size` - The size of the new ScopedMemory area in bytes.

`logic` - The Runnable to execute when this ScopedMemory is entered. If `logic` is null, this constructor is equivalent to constructing the memory area without a logic value.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`IllegalAssignmentError448` - Thrown if storing `logic` in this would violate the assignment rules.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the ScopedMemory object or for the backing memory.

```
public ScopedMemory(javax.realtime.SizeEstimator170 size)
```

Create a new ScopedMemory area with the given parameters.

Parameters:

`size` - The size of the new ScopedMemory area estimated by an instance of `SizeEstimator170`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the ScopedMemory object or for the backing memory.

```
public ScopedMemory(
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create a new ScopedMemory area with the given parameters.

Parameters:

`size` - The size of the new ScopedMemory area estimated by an instance of `SizeEstimator170`.

`logic` - The logic which will use the memory represented by `this` as its initial memory area. If `logic` is null, this constructor is equivalent to constructing the memory area without a logic value.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ScopedMemory` object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing `logic` in this would violate the assignment rules.

7.5.2 Methods

`public void enter()`

Associate this memory area with the current schedulable object for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea(Runnable)`₁₇₇) or the `enter` method exits.

Overrides: `enter`₁₆₃ in class `MemoryArea`₁₆₁

Throws:

`ScopedCycleException`₄₅₅ - Thrown if this invocation would break the single parent rule.

`ThrowBoundaryError`₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`₄₄₈, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`₄₅₆ is allocated in the current allocation context and contains information about the exception it replaces.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization.

This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and no non-null value for `logic` was supplied when the memory area was constructed.

`MemoryAccessError`₄₅₀ - Thrown if caller is a no-heap schedulable object and this memory area's `logic` value is allocated in heap memory.

public void enter(`java.lang.Runnable logic`)

Associate this memory area with the current schedulable object for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea(Runnable)`₁₇₇) or the `enter` method exits.

Overrides: `enter`₁₆₄ in class `MemoryArea`₁₆₁

Parameters:

`logic` - The `Runnable` object whose `run()` method should be invoked.

Throws:

`ScopedCycleException`₄₅₅ - Thrown if this invocation would break the single parent rule.

`ThrowBoundaryError`₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`₄₄₈, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`₄₅₆ is allocated in the current allocation context and contains information about the exception it replaces.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and `logic` is null.

```
public void executeInArea(java.lang.Runnable logic)
```

Execute the run method from the `logic` parameter using this memory area as the current allocation context. This method behaves as if it moves the allocation context down the scope stack to the occurrence of this.

Overrides: [executeInArea₁₆₅](#) in class [MemoryArea₁₆₁](#)

Parameters:

`logic` - The runnable object whose `run()` method should be executed.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

[InaccessibleAreaException₄₄₉](#) - Thrown if the memory area is not in the schedulable object's scope stack.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and `logic` is null.

```
public long getMaximumSize()
```

Get the maximum size this memory area can attain. If this is a fixed size memory area, the returned value will be equal to the initial size.

Returns: The maximum size attainable.

```
public java.lang.Object getPortal()
```

Return a reference to the portal object in this instance of `ScopedMemory`.

Assignment rules are enforced on the value returned by `getPortal` as if the return value were first stored in an object allocated in the current allocation context, then moved to its final destination.

Returns: A reference to the portal object or null if there is no portal object. The portal value is always set to null when the contents of the memory are deleted.

Throws:

[IllegalAssignmentError₄₄₈](#) - Thrown if a reference to the portal object cannot be stored in the caller's allocation context; that is, if this is "inner" relative to the current allocation context or not on the caller's scope stack.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

public int getReferenceCount()

Returns the reference count of this `ScopedMemory`.

Note: A reference count of 0 reliably means that the scope is not referenced, but other reference counts are subject to artifacts of lazy/eager maintenance by the implementation.

Returns: The reference count of this `ScopedMemory`.

public void join()
throws `InterruptedException`

Wait until the reference count of this `ScopedMemory` goes down to zero. Return immediately if the memory is unreferenced.

Throws:

`java.lang.InterruptedException` - If this schedulable object is interrupted by `RealtimeThread.interrupt()`³⁶ or `AsynchronouslyInterruptedException.fire()`⁴²⁵ while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

public void join(
`javax.realtime.HighResolutionTime`₃₁₄ `time)
throws InterruptedException`

Wait at most until the time designated by the `time` parameter for the reference count of this `ScopedMemory` to drop to zero. Return immediately if the memory area is unreferenced.

Since the time is expressed as a `HighResolutionTime`₃₁₄, this method is an accurate timer with nanosecond granularity. The actual resolution of the timer and even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. If relative, then the delay is the amount of time given by `time`, and measured by its associated clock. If absolute, then the delay is until the indicated value is reached by the clock. If the given absolute time is less than or equal to the current value of the clock, the call to `join` returns immediately.

Parameters:

`time` - If this time is an absolute time, the wait is bounded by that point in time. If the time is a relative time (or a member of the `RationalTime` subclass of `RelativeTime`) the wait is bounded by a the specified interval from some time between the time

`join` is called and the time it starts waiting for the reference count to reach zero.

Throws:

`java.lang.InterruptedException` - If this schedulable object is interrupted by `RealtimeThread.interrupt()`³⁶ or `AsynchronouslyInterruptedException.fire()`⁴²⁵ while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and time is null.

`java.lang.UnsupportedOperationException` - Thrown if the wait operation is not supported using the clock associated with time.

```
public void joinAndEnter()
    throws InterruptedException
```

In the error-free case, `joinAndEnter` combines `join()`; `enter()`; such that no `enter()` from another schedulable object can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, then enter the `ScopedMemory` and execute the `run` method from `logic` passed in the constructor. If no instance of `java.lang.Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately.

If multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable object is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Throws:

`java.lang.InterruptedException` - If this schedulable object is interrupted by `RealtimeThread.interrupt()`³⁶ or `AsynchronouslyInterruptedException.fire()`⁴²⁵ while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization

of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

ThrowBoundaryError₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an **IllegalAssignmentError₄₄₈**, so the JVM cannot be permitted to deliver the exception. The **ThrowBoundaryError₄₅₆** is allocated in the current allocation context and contains information about the exception it replaces.

ScopedCycleException₄₅₅ - Thrown if this invocation would break the single parent rule.

java.lang.IllegalArgumentException - Thrown if the caller is a schedulable object and no non-null `logic` value was supplied to the memory area's constructor.

MemoryAccessError₄₅₀ - Thrown if caller is a non-heap schedulable object and this memory area's `logic` value is allocated in heap memory.

```
public void joinAndEnter(
    javax.realtime.HighResolutionTime314 time)
    throws InterruptedException
```

In the error-free case, `joinAndEnter` combines `join()`; `enter()`; such that no `enter()` from another schedulable object can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, or for the current time to reach the designated time, then enter the `ScopedMemory` and execute the `run` method from `Runnable` object passed to the constructor. If no instance of `java.lang.Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately. *

If multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Since the time is expressed as a **HighResolutionTime₃₁₄**, this method has an accurate timer with nanosecond granularity. The actual resolution of the timer and even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. If relative, then the calling thread is blocked for at most the amount of time given by `time`, and

measured by its associated clock. If absolute, then the time delay is until the indicated value is reached by the clock. If the given absolute time is less than or equal to the current value of the clock, the call to `joinAndEnter` returns immediately.

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable object is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

Parameters:

`time` - The time that bounds the wait.

Throws:

[ThrowBoundaryError₄₅₆](#) - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an [IllegalAssignmentError₄₄₈](#), so the JVM cannot be permitted to deliver the exception. The [ThrowBoundaryError₄₅₆](#) is allocated in the current allocation context and contains information about the exception it replaces.

`java.lang.InterruptedException` - If this schedulable object is interrupted by [RealtimeThread.interrupt\(\)₃₆](#) or [AsynchronouslyInterruptedException.fire\(\)₄₂₅](#) while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

[ScopedCycleException₄₅₅](#) - Thrown if the caller is a schedulable object and this invocation would break the single parent rule.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object, and `time` is null or no non-null logic value was supplied to the memory area's constructor.

`java.lang.UnsupportedOperationException` - Thrown if the wait operation is not supported using the clock associated with `time`.

[MemoryAccessError](#)₄₅₀ - Thrown if caller is a no-heap schedulable object and this memory area's logic value is allocated in heap memory.

```
public void joinAndEnter(java.lang.Runnable logic)
    throws InterruptedException
```

In the error-free case, `joinAndEnter` combines `join()`; `enter()`; such that no `enter()` from another schedulable object can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, then enter the `ScopedMemory` and execute the run method from `logic`

If `logic` is null, throw `IllegalArgumentException` immediately.

If multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable object is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Parameters:

`logic` - The `java.lang.Runnable` object which contains the code to execute.

Throws:

`java.lang.InterruptedException` - If this schedulable object is interrupted by [RealtimeThread.interrupt\(\)](#)₃₆ or [AsynchronouslyInterruptedException.fire\(\)](#)₄₂₅ while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

[ThrowBoundaryError](#)₄₅₆ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an [IllegalAssignmentError](#)₄₄₈, so the JVM cannot be permitted to deliver the exception. The

`ThrowBoundaryError456` is allocated in the current allocation context and contains information about the exception it replaces.

`ScopedCycleException455` - Thrown if this invocation would break the single parent rule.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and `logic` is null.

```
public void joinAndEnter(
    java.lang.Runnable logic,
    javax.realtime.HighResolutionTime314 time)
    throws InterruptedException
```

In the error-free case, `joinAndEnter` combines `join()`; `enter()`; such that no `enter()` from another schedulable object can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, or for the current time to reach the designated time, then enter the `ScopedMemory` and execute the run method from `logic`.

Since the time is expressed as a `HighResolutionTime314`, this method is an accurate timer with nanosecond granularity. The actual resolution of the timer and even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. If relative, then the delay is the amount of time given by `time`, and measured by its associated clock. If absolute, then the delay is until the indicated value is reached by the clock. If the given absolute time is less than or equal to the current value of the clock, the call to `join` returns immediately.

Throws `IllegalArgumentException` immediately if `logic` is null.

If multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable object is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

Parameters:

`logic` - The `java.lang.Runnable` object which contains the code to execute.

`time` - The time that bounds the wait.

Throws:

`java.lang.InterruptedException` - If this schedulable object is interrupted by `RealtimeThread.interrupt()`³⁶ or `AsynchronouslyInterruptedException.fire()`⁴²⁵ while waiting for the reference count to go to zero.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread, or if this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

`ThrowBoundaryError`⁴⁵⁶ - Thrown when the JVM needs to propagate an exception allocated in this scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`⁴⁴⁸, so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`⁴⁵⁶ is allocated in the current allocation context and contains information about the exception it replaces.

`ScopedCycleException`⁴⁵⁵ - Thrown if the caller is a schedulable object and this invocation would break the single parent rule.

`java.lang.IllegalArgumentException` - Thrown if the caller is a schedulable object and `time` or `logic` is null.

`java.lang.UnsupportedOperationException` - Thrown if the wait operation is not supported using the clock associated with `time`.

```
public java.lang.Object newArray(
    java.lang.Class type,
    int number)
```

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Overrides: `newArray`¹⁶⁵ in class `MemoryArea`¹⁶¹

Parameters:

`type` - The class of the elements of the new array. To create an array of a primitive type use a type such as `Integer.TYPE` (which would call for an array of the primitive `int` type.)

`number` - The number of elements in the new array.

Returns: A new array of class type, of number elements.

Throws:

`java.lang.IllegalArgumentException` - Thrown if number is less than zero, type is null, or type is `java.lang.Void.TYPE`.

`java.lang.OutOfMemoryError` - Thrown if space in the memory area is exhausted.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

[InaccessibleAreaException₄₄₉](#) - Thrown if the memory area is not in the schedulable object's scope stack.

```
public java.lang.Object newInstance(java.lang.Class type)
    throws IllegalAccessException,
    InstantiationException
```

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Overrides: [newInstance₁₆₆](#) in class [MemoryArea₁₆₁](#)

Parameters:

type - The class of which to create a new instance.

Returns: A new instance of class type.

Throws:

`java.lang.IllegalAccessException` - The class or initializer is inaccessible.

`java.lang.IllegalArgumentException` - Thrown if type is null.

`java.lang.ExceptionInInitializerError` - Thrown if an unexpected exception has occurred in a static initializer

`java.lang.OutOfMemoryError` - Thrown if space in the memory area is exhausted.

`java.lang.InstantiationException` - Thrown if the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array, or an exception was thrown by the constructor.

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java thread.

[InaccessibleAreaException₄₄₉](#) - Thrown if the memory area is not in the schedulable object's scope stack.

```
public java.lang.Object newInstance(
    java.lang.reflect.Constructor c,
    java.lang.Object[] args)
    throws IllegalAccessException,
    InstantiationException, InvocationTargetException
```

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Overrides: [newInstance₁₆₇](#) in class [MemoryArea₁₆₁](#)

Parameters:

c -

args - An array of arguments to pass to the constructor.

Returns: A new instance of the object constructed by *c*.

Throws:

[java.lang.IllegalAccessException](#) - Thrown if the class or initializer is inaccessible under Java access control.

[java.lang.InstantiationException](#) - Thrown if the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array.

[java.lang.OutOfMemoryError](#) - Thrown if space in the memory area is exhausted.

[java.lang.IllegalArgumentException](#) - Thrown if *c* is null, or the *args* array does not contain the number of arguments required by *c*. A null value of *args* is treated like an array of length 0.

[java.lang.IllegalThreadStateException](#) - Thrown if the caller is a Java thread.

[java.lang.reflect.InvocationTargetException](#) - Thrown if the underlying constructor throws an exception.

[InaccessibleAreaException₄₄₉](#) - Thrown if the memory area is not in the schedulable object's scope stack.

```
public void setPortal(java.lang.Object object)
```

Sets the *portal* object of the memory area represented by this instance of [ScopedMemory](#) to the given object. The object must have been allocated in this [ScopedMemory](#) instance.

Parameters:

`object` - The object which will become the portal for this. If null the previous portal object remains the portal object for this or if there was no previous portal object then there is still no portal object for this.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if the caller is a Java Thread, and `object` is not null.

`IllegalAssignmentError`₄₄₈ - Thrown if the caller is a schedulable object, and `object` is not allocated in this scoped memory instance and not null.

`InaccessibleAreaException`₄₄₉ - Thrown if the caller is a schedulable object, this memory area is not in the caller's scope stack and `object` is not null.

```
public java.lang.String toString()
```

Returns a user-friendly representation of this `ScopedMemory` of the form `Scoped memory # <num>` where `<num>` is a number that uniquely identifies this scoped memory area.

Overrides: `toString` in class `Object`

Returns: The string representation

7.6 LTMemory

Declaration

```
public class LTMemory extends ScopedMemory172
```

Description

`LTMemory` represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the memory area's *initial* size. Execution time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available.

The memory area described by a `LTMemory` instance does not exist in the Java heap, and is not subject to garbage collection. Thus, it is safe to use a `LTMemory` object as the initial memory area associated with a `NoHeapRealtimeThread`₅₅, or to enter

the memory area using the `ScopedMemory.enter()`₁₇₅ method within a `NoHeapRealtimeThread`₅₅.

Enough memory must be committed by the completion of the constructor to satisfy the initial memory requirement. (Committed means that this memory must always be available for allocation). The initial memory allocation must behave, with respect to successful allocation, as if it were contiguous; i.e., a correct implementation must guarantee that any sequence of object allocations that could ever succeed without exceeding a specified initial memory size will always succeed without exceeding that initial memory size and succeed for any instance of `LMemory` with that initial memory size.

(Note: to ensure that all requested memory is available set initial and maximum to the same value)

Methods from `LMemory` should be overridden only by methods that use `super`.

See Also: `MemoryArea`₁₆₁, `ScopedMemory`₁₇₂, `RealtimeThread`₂₉, `NoHeapRealtimeThread`₅₅

7.6.1 Constructors

```
public LMemory(long size)
```

Create an `LMemory` of the given size. This constructor is equivalent to `LMemory(size, size)`

Parameters:

`size` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `LMemory` object or for the backing memory.

Since: 1.0.1

```
public LMemory(
    long initial,
    long maximum)
```

Create an `LMemory` of the given size.

Parameters:

`initial` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

`maximum` - The size in bytes of the memory to allocate for this area.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is greater than `maximum`, or if `initial` or `maximum` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `LTMemory` object or for the backing memory.

```
public LTMemory(
    long initial,
    long maximum,
    java.lang.Runnable logic)
```

Create an `LTMemory` of the given size.

Parameters:

`initial` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

`maximum` - The size in bytes of the memory to allocate for this area.

`logic` - The `run()` of the given `Runnable` will be executed using this as its initial memory area. If `logic` is null, this constructor is equivalent to `LTMemory(long, long)`¹⁸⁸.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is greater than `maximum`, or if `initial` or `maximum` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `LTMemory` object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing `logic` in this would violate the assignment rules.

```
public LMemory(
    long size,
    java.lang.Runnable logic)
```

Create an LMemory of the given size. This constructor is equivalent to LMemory(size, size, logic).

Parameters:

size - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

logic - The run() of the given Runnable will be executed using this as its initial memory area. If logic is null, this constructor is equivalent to LMemory(long)₁₈₈.

Throws:

java.lang.IllegalArgumentException - Thrown if size is less than zero.

java.lang.OutOfMemoryError - Thrown if there is insufficient memory for the LMemory object or for the backing memory.

IllegalAssignmentError₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

Since: 1.0.1

```
public LMemory(javax.realtime.SizeEstimator170 size)
```

Create an LMemory of the given size. This constructor is equivalent to LMemory(size, size).

Parameters:

size - An instance of SizeEstimator₁₇₀ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

Throws:

java.lang.IllegalArgumentException - Thrown if size is null, or size.getEstimate() is less than zero.

java.lang.OutOfMemoryError - Thrown if there is insufficient memory for the LMemory object or for the backing memory.

Since: 1.0.1

```
public LMemory(
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an LMemory of the given size.

Parameters:

`size` - An instance of `SizeEstimator170` used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`logic` - The `run()` of the given `Runnable` will be executed using this as its initial memory area. If `logic` is null, this constructor is equivalent to `LMemory(SizeEstimator190)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the LMemory object or for the backing memory.

`IllegalAssignmentError448` - Thrown if storing `logic` in this would violate the assignment rules.

Since: 1.0.1

```
public LMemory(
    javax.realtime.SizeEstimator170 initial,
    javax.realtime.SizeEstimator170 maximum)
```

Create an LMemory of the given size.

Parameters:

`initial` - An instance of `SizeEstimator170` used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`maximum` - An instance of `SizeEstimator170` used to give an estimate for the maximum bytes to allocate for this area.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or if `initial.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the LMemory object or for the backing memory.

```
public LMemory(
    javax.realtime.SizeEstimator170 initial,
    javax.realtime.SizeEstimator170 maximum,
    java.lang.Runnable logic)
```

Create an LMemory of the given size.

Parameters:

`initial` - An instance of `SizeEstimator170` used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`maximum` - An instance of `SizeEstimator170` used to give an estimate for the maximum bytes to allocate for this area.

`logic` - The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. If `logic` is null, this constructor is equivalent to `LMemory(SizeEstimator, SizeEstimator)191`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or if `initial.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the LMemory object or for the backing memory.

`IllegalAssignmentError448` - Thrown if storing `logic` in this would violate the assignment rules.

7.6.2 Methods

```
public java.lang.String toString()
```

Create a string representation of this object. The string is of the form
(LMemory) Scoped memory # num

where `num` uniquely identifies the LMemory area.

Overrides: `toString187` in class `ScopedMemory172`

Returns: A string representing the value of `this`.

7.7 VTMemory

Declaration

```
public class VTMemory extends ScopedMemory172
```

Description

VTMemory is similar to [LTMemory₁₈₇](#) except that the execution time of an allocation from a VTMemory area need not complete in linear time.

Methods from VTMemory should be overridden only by methods that use super.

7.7.1 Constructors

```
public VTMemory(long size)
```

Create an VTMemory of the given size. This constructor is equivalent to VTMemory(size, size)

Parameters:

size - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

Throws:

java.lang.IllegalArgumentException - Thrown if size is less than zero.

java.lang.OutOfMemoryError - Thrown if there is insufficient memory for the VTMemory object or for the backing memory.

Since: 1.0.1

```
public VTMemory(long initial, long maximum)
```

Creates a VTMemory with the given parameters.

Parameters:

initial - The size in bytes of the memory to initially allocate for this area.

maximum - The maximum size in bytes this memory area to which the size may grow.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is greater than `maximum` or if `initial` or `maximum` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

```
public VTMemory(
    long initial,
    long maximum,
    java.lang.Runnable logic)
```

Creates a `VTMemory` with the given parameters.

Parameters:

`initial` - The size in bytes of the memory to initially allocate for this area.

`maximum` - The maximum size in bytes this memory area to which the size may grow.

`logic` - An instance of `java.lang.Runnable` whose `run()` method will use this as its initial memory area. If `logic` is null, this constructor is equivalent to `VTMemory(long, long)`¹⁹³.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is greater than `maximum`, or if `initial` or `maximum` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

`IllegalAssignmentError`⁴⁴⁸ - Thrown if storing `logic` in this would violate the assignment rules.

```
public VTMemory(long size, java.lang.Runnable logic)
```

Create an `VTMemory` of the given size. This constructor is equivalent to `VTMemory(size, size, logic)`.

Parameters:

`size` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

`logic` - The `run()` of the given `Runnable` will be executed using this as its initial memory area. If `logic` is null, this constructor is equivalent to `VTMemory(long)`¹⁹³ .

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

`IllegalAssignmentError`⁴⁴⁸ - Thrown if storing `logic` in this would violate the assignment rules.

Since: 1.0.1

```
public VTMemory(javax.realtime.SizeEstimator170 size)
```

Create an `VTMemory` of the given size. This constructor is equivalent to `VTMemory(size, size)`.

Parameters:

`size` - An instance of `SizeEstimator`₁₇₀ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

Since: 1.0.1

```
public VTMemory(  
    javax.realtime.SizeEstimator170 size,  
    java.lang.Runnable logic)
```

Create an `VTMemory` of the given size.

Parameters:

`size` - An instance of `SizeEstimator`₁₇₀ used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`logic` - The `run()` of the given `Runnable` will be executed using this as its initial memory area. If `logic` is null, this constructor is equivalent to `VTMemory(SizeEstimator)`¹⁹⁵ .

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

Since: 1.0.1

```
public VTMemory(
    javax.realtime.SizeEstimator170 initial,
    javax.realtime.SizeEstimator170 maximum)
```

Creates a `VTMemory` with the given parameters.

Parameters:

`initial` - The size in bytes of the memory to initially allocate for this area.

`maximum` - The maximum size in bytes this memory area to which the size may grow estimated by an instance of `SizeEstimator`₁₇₀.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or if `initial.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

```
public VTMemory(
    javax.realtime.SizeEstimator170 initial,
    javax.realtime.SizeEstimator170 maximum,
    java.lang.Runnable logic)
```

Creates a `VTMemory` with the given parameters.

Parameters:

`initial` - The size in bytes of the memory to initially allocate for this area.

maximum - The maximum size in bytes this memory area to which the size may grow estimated by an instance of [SizeEstimator](#)₁₇₀.

logic - An instance of `java.lang.Runnable` whose `run()` method will use this as its initial memory area. If `logic` is null, this constructor is equivalent to [VTMemory\(SizeEstimator, SizeEstimator\)](#)₁₉₆.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `initial` is null, `maximum` is null, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or if `initial.getEstimate()` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `VTMemory` object or for the backing memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing `logic` in this would violate the assignment rules.

7.7.2 Methods

```
public java.lang.String toString()
```

Create a string representing this object. The string is of the form
(VTMemory) Scoped memory # num

where `num` uniquely identifies the `VTMemory` area.

Overrides: [toString](#)₁₈₇ in class [ScopedMemory](#)₁₇₂

Returns: A string representing the value of `this`.

7.8 PhysicalMemoryManager

Declaration

```
public final class PhysicalMemoryManager
```

Description

The `PhysicalMemoryManager` is not ordinarily used by applications, except that the implementation may require the application to use the [registerFilter\(Object, PhysicalMemoryTypeFilter\)](#)₂₀₃ method to make the physical memory manager aware of the memory types on their platform. The `PhysicalMemoryManager` class is

primarily intended for use by the various physical memory accessor objects ([VTPhysicalMemory₂₃₀](#), [LTPhysicalMemory₂₂₁](#), [ImmortalPhysicalMemory₂₁₂](#), [RawMemoryAccess₂₃₉](#) and [RawMemoryFloatAccess₂₆₂](#)) to create objects of the types requested by the application. The physical memory manager is responsible for finding areas of physical memory with the appropriate characteristics and access rights, and moderating any required combination of physical and virtual memory characteristics.

Examples of characteristics that might be specified are: DMA memory, hardware byte swapping, non-cached access to memory, etc. Standard “names” for some memory characteristics are included in this class — DMA, SHARED, ALIGNED, BYTESWAP, and IO_PAGE — support for these characteristics is optional, but if they are supported they must use these names. Additional characteristics may be supported, but only names defined in this specification may be visible in the `PhysicalMemoryManager` API.

The base implementation will provide a `PhysicalMemoryManager`.

Original Equipment Manufacturers or other interested parties may provide [PhysicalMemoryTypeFilter₂₀₅](#) classes that allow additional characteristics of memory devices to be specified.

Memory attributes that are configured may not be compatible with one another. For instance, copy-back cache enable may be incompatible with execute-only. In this case, the implementation of memory filters may detect conflicts and throw a [MemoryTypeConflictException₄₅₂](#), but since filters are not part of the normative RTSJ, this exception is only advisory.

7.8.1 Fields

```
public static final java.lang.Object ALIGNED
```

If aligned memory is supported by the implementation specify ALIGNED to identify aligned memory.

```
public static final java.lang.Object BYTESWAP
```

If automatic byte swapping is supported by the implementation specify BYTESWAP if byte swapping should be used.

```
public static final java.lang.Object DMA
```

If DMA memory is supported by the implementation, specify DMA to identify DMA memory.

```
public static final java.lang.Object IO_PAGE
```

If access to the system I/O space is supported by the implementation specify `IO_PAGE` if I/O space should be used.

Since: 1.0.1

```
public static final java.lang.Object SHARED
```

If shared memory is supported by the implementation specify `SHARED` to identify shared memory.

7.8.2 Methods

```
public static boolean isRemovable(
    long base,
    long size)
```

Queries the system about the removability of the specified range of memory.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

Returns: true if any part of the specified range can be removed.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`SizeOutOfBoundsException455` - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

`OffsetOutOfBoundsException454` - Thrown if `base` is less than zero.

```
public static boolean isRemoved(long base, long size)
```

Queries the system about the removed state of the specified range of memory. This method is used for devices that lie in the memory address space and can be removed while the system is running. (Such as PC cards).

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

Returns: true if any part of the specified range is currently not usable.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`OffsetOutOfBoundsException454` - Thrown if `base` is less than zero.

`SizeOutOfBoundsException455` - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

```
public static void onInsertion(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Register the specified `AsyncEvent388` to fire when any memory in the range is added to the system. If the specified range of physical memory contains multiple different types of removable memory, the AE will be registered with each of them.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

`ae` - The async event to fire.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `ae` is null, or if the specified range contains no removable memory, or if `size` is less than zero.

`OffsetOutOfBoundsException454` - Thrown if `base` is less than zero.

`SizeOutOfBoundsException455` - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public static void onInsertion(
    long base,
    long size,
    javax.realtime.AsyncEventHandler393 aeh)
```

Deprecated. 1.0.1 Replace with onInsertion(long, long, AsyncEvent)

Register the specified [AsyncEventHandler³⁹³](#) to run when any memory in the range is added to the system. If the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of them. If the size or the base is less than 0, unregister all “onInsertion” references to the handler.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

`aeh` - The handler to register.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `aeh` is null, or if the specified range contains no removable memory, or if `aeh` is null and `size` and `base` are both greater than or equal to zero.

[SizeOutOfBoundsException⁴⁵⁵](#) - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

```
public static void onRemoval(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Register the specified AE to fire when any memory in the range is removed from the system. If the specified range of physical memory contains multiple different types of removable memory, the AE will be registered with each of them.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

ae - The async event to register.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the specified range contains no removable memory, if ae is null, or if size is less than zero.

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public static void onRemoval(
    long base,
    long size,
    javax.realtime.AsyncEventHandler393 aeh)
```

Deprecated. 1.0.1

Register the specified AEH to run when any memory in the range is removed from the system. If the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of them. If size or base is less than 0, unregister all “onRemoval” references to the handler parameter.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

Parameters:

base - The starting address in physical memory.

size - The size of the memory area.

ae - The handler to register.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the specified range contains no removable memory, or if aeh is null and size and base are both greater than or equal to zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

```
public static final void registerFilter(
    java.lang.Object name,
    javax.realtime.PhysicalMemoryTypeFilter205 filter)
    throws DuplicateFilterException
```

Register a memory type filter with the physical memory manager.

Values of name are compared using reference equality (==) not value equality (equals()).

Parameters:

name - The type of memory handled by this filter.

filter - The filter object.

Throws:

[DuplicateFilterException₄₄₈](#) - Thrown if a filter for this type of memory already exists.

[ResourceLimitError₄₅₈](#) - Thrown if the system is configured for a bounded number of filters. This filter exceeds the bound.

[java.lang.IllegalArgumentException](#) - Thrown if the name parameter is an array of objects, if the name and filter are not both in immortal memory, or if either name or filter is null.

[java.lang.SecurityException](#) - Thrown if this operation is not permitted.

```
public static final void removeFilter(
    java.lang.Object name)
```

Remove the identified filter from the set of registered filters. If the filter is not registered, silently do nothing.

Values of name are compared using reference equality (==) not value equality (equals()).

Parameters:

name - The identifying object for this memory attribute.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if name is null.

[java.lang.SecurityException](#) - Thrown if this operation is not permitted.

```
public static boolean unregisterInsertionEvent(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Unregister the specified insertion event. The event is only unregistered if all three arguments match the arguments used to register the event, except that ae of null matches all values of ae and will unregister every ae that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Parameters:

base - The starting address in physical memory associated with ae.

size - The size of the memory area associated with ae.

ae - The event to unregister.

Returns: True if at least one event matched the pattern, false if no such event was found.

Throws:

`java.lang.IllegalArgumentException` - Thrown if size is less than 0.

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public static boolean unregisterRemovalEvent(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Unregister the specified removal event. The async event is only unregistered if all three arguments match the arguments used to register the event, except that ae of null matches all values of ae and will unregister every ae that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Parameters:

`base` - The starting address in physical memory associated with `ae`.

`size` - The size of the memory area associated with `ae`.

`ae` - The async event to unregister.

Returns: True if at least one event matched the pattern, false if no such event was found.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than 0.

`OffsetOutOfBoundsException454` - Thrown if `base` is less than zero.

`SizeOutOfBoundsException455` - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

Since: 1.0.1

7.9 PhysicalMemoryTypeFilter

Declaration

```
public interface PhysicalMemoryTypeFilter
```

Description

Implementation or device providers may include classes that implement `PhysicalMemoryTypeFilter` which allow additional characteristics of memory in devices to be specified. Implementations of `PhysicalMemoryTypeFilter` are intended to be used by the `PhysicalMemoryManager197`, not directly from application code.

7.9.1 Methods

```
public boolean contains(long base, long size)
```

Queries the system about whether the specified range of memory contains any of this type.

Parameters:

`base` - The physical address of the beginning of the memory region.

`size` - The size of the memory region.

Returns: true if the specified range contains ANY of this type of memory.

Throws:

`java.lang.IllegalArgumentException` - Thrown if base or size is negative.

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

See Also: `PhysicalMemoryManager.isRemovable(long, long)199`

```
public long find(long base, long size)
```

Search for physical memory of the right type.

Parameters:

base - The physical address at which to start searching.

size - The amount of memory to be found.

Returns: The address where memory was found or -1 if it was not found.

Throws:

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

`java.lang.IllegalArgumentException` - Thrown if base or size is negative.

```
public int getVMAttributes()
```

Gets the virtual memory attributes of `this`. The value of this field is as defined for the POSIX `mmap` function's `prot` parameter for the platform. The meaning of the bits is platform-dependent. POSIX defines constants for `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, and `PROT_NONE`.

Returns: The virtual memory attributes as an integer.

```
public int getVMFlags()
```

Gets the virtual memory flags of `this`. The value of this field is as defined for the POSIX `mmap` function's `flags` parameter for the platform. The

meaning of the bits is platform-dependent. POSIX defines constants for MAP_SHARED, MAP_PRIVATE, and MAP_FIXED.

Returns: The virtual memory flags as an integer.

public void initialize(long base, long vBase, long size)

If configuration is required for memory to fit the attribute of this object, do the configuration here.

Parameters:

base - The address of the beginning of the physical memory region.

vBase - The address of the beginning of the virtual memory region.

size - The size of the memory region.

Throws:

`java.lang.IllegalArgumentException` - Thrown if **base** or **size** is negative.

`OffsetOutOfBoundsException454` - Thrown if **base** is less than zero.

`SizeOutOfBoundsException455` - Thrown if **base** plus **size** would be greater than the physical addressing range of the processor, or **vBase** plus **size** would exceed the virtual addressing range of the processor.

public boolean isPresent(long base, long size)

Queries the system about the existence of the specified range of physical memory.

Parameters:

base - The address of the beginning of the memory region.

size - The size of the memory region.

Returns: True if all of the memory is present. False if any of the memory has been removed.

Throws:

`java.lang.IllegalArgumentException` - if the **base** and **size** do not fall into this type of memory.

`OffsetOutOfBoundsException454` - Thrown if **base** is less than zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if base plus size would be greater than the physical addressing range of the processor.

See Also: [PhysicalMemoryManager.isRemoved\(long, long\)₁₉₉](#)

```
public boolean isRemovable()
```

Queries the system about the removability of this memory.

Returns: true if this type of memory is removable.

```
public void onInsertion(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Register the specified [AsyncEvent₃₈₈](#) to fire when any memory of this type in the range is added to the system.

Parameters:

base - The starting address in physical memory.

size - The size of the memory area.

ae - The async event to fire.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if ae is null, or if the specified range contains no removable memory of this type. [IllegalArgumentException](#) may also be thrown if size is less than zero.

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if base is less than zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if base plus size would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public void onInsertion(long base, long size,
    javax.realtime.AsyncEventHandler393 aeh)
```

Deprecated. 1.0.1 Replace with [onInsertion\(long, long, AsyncEvent\)](#)

Register the specified [AsyncEventHandler₃₉₃](#) to run when any memory of this type, and in the range is added to the system. If the size or the base is less than 0, unregister all “onInsertion” references to the handler.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

Parameters:

base - The starting address in physical memory.

size - The size of the memory area.

aeh - The handler to register.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if the specified range contains no removable memory, or if aeh is null and size and base are both greater than or equal to zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if base plus size would be greater than the physical addressing range of the processor.

```
public void onRemoval(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Register the specified AE to fire when any memory in the range is removed from the system.

Parameters:

base - The starting address in physical memory.

size - The size of the memory area.

ae - The async event to register.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if the specified range contains no removable memory of this type, if ae is null, or if size is less than zero.

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if base is less than zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public void onRemoval(
    long base,
    long size,
    javax.realtime.AsyncEventHandler393 aeh)
```

Deprecated. 1.0.1

Register the specified AEH to run when any memory in the range is removed from the system. If `size` or `base` is less than 0, unregister all “onRemoval” references to the handler parameter.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

`aeh` - The handler to register.

Throws:

[java.lang.IllegalArgumentException](#) - Thrown if the specified range contains no removable memory known to this filter, if `aeh` is null and `size` and `base` are both greater than or equal to zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

```
public boolean unregisterInsertionEvent(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Unregister the specified insertion event. The event is only unregistered if all three arguments match the arguments used to register the event, except that `ae` of null matches all values of `ae` and will unregister every `ae` that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Parameters:

`base` - The starting address in physical memory associated with `ae`.

`size` - The size of the memory area associated with `ae`.

`ae` - The event to unregister.

Returns: True if at least one event matched the pattern, false if no such event was found.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than 0.

`OffsetOutOfBoundsException454` - Thrown if `base` is less than zero.

`SizeOutOfBoundsException455` - Thrown if `base` plus `size` would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public boolean unregisterRemovalEvent(
    long base,
    long size,
    javax.realtime.AsyncEvent388 ae)
```

Unregister the specified removal event. The async event is only unregistered if all three arguments match the arguments used to register the event, except that `ae` of null matches all values of `ae` and will unregister every `ae` that matches the address range. Note: This method has no effect on handlers registered directly as async event handlers.

Parameters:

`base` - The starting address in physical memory associated with `ae`.

`size` - The size of the memory area associated with `ae`.

`ae` - The async event to unregister.

Returns: True if at least one event matched the pattern, false if no such event was found.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `size` is less than 0.

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

Since: 1.0.1

```
public long vFind(long base, long size)
```

Search for virtual memory of the right type. This is important for systems where attributes are associated with particular ranges of virtual memory.

Parameters:

base - The address at which to start searching.

size - The amount of memory to be found.

Returns: The address where memory was found or -1 if it was not found.

Throws:

`OffsetOutOfBoundsException454` - Thrown if base is less than zero.

`SizeOutOfBoundsException455` - Thrown if base plus size would be greater than the physical addressing range of the processor.

`java.lang.IllegalArgumentException` - Thrown if base or size is negative. `IllegalArgumentException` may also be thrown if base is an invalid virtual address.

7.10 ImmortalPhysicalMemory

Declaration

```
public class ImmortalPhysicalMemory extends MemoryArea161
```

Description

An instance of `ImmortalPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same restrictive set of assignment rules as `ImmortalMemory168` memory areas, and may be used in any execution context where `ImmortalMemory` is appropriate.

No provision is made for sharing object in `ImmortalPhysicalMemory` with entities outside the JVM that creates them, and, while the memory backing an instance of `ImmortalPhysicalMemory` could be shared by multiple JVMs, the class does not support such sharing.

Methods from `ImmortalPhysicalMemory` should be overridden only by methods that use `super`.

7.10.1 Constructors

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long size)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`UnsupportedPhysicalMemoryException`₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`₂₀₅ has been registered with the `PhysicalMemoryManager`₁₉₇.

`MemoryTypeConflictException`₄₅₂ - Thrown if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the `size` extends into an invalid range of memory.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long base,
    long size)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the base address is invalid.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the `size` extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero. `IllegalArgumentException` may also be thrown if `base` plus `size` would be greater than the maximum physical address supported by the processor.

[MemoryInUseException](#)₄₅₀ - Thrown if the specified memory is already in use.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long base,
    long size,
    java.lang.Runnable logic)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)](#)₁₆₃ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the base address is invalid.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if `size` extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇.

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if size is negative. `IllegalArgumentException` may also be thrown if base plus size would be greater than the maximum physical address supported by the processor.

[MemoryInUseException](#)₄₅₀ - Thrown if the specified memory is already in use.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long size,
    java.lang.Runnable logic)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)](#)₁₆₃ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if size extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇.

`java.lang.IllegalArgumentException` - Thrown if size is negative.

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the base address is invalid.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the size estimate from `size` extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`IllegalArgumentException` may also be thrown if base plus the size indicated by `size` would be greater than the maximum physical address supported by the processor.

[MemoryInUseException₄₅₀](#) - Thrown if the specified memory is already in use.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁶³ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`OffsetOutOfBoundsException`⁴⁵⁴ - Thrown if the base address is invalid.

`SizeOutOfBoundsException`⁴⁵⁵ - Thrown if the size estimate from `size` extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`⁴⁵⁷ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`²⁰⁵ has been registered with the `PhysicalMemoryManager`¹⁹⁷.

`MemoryTypeConflictException`⁴⁵² - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`IllegalArgumentException` may also be thrown if `base` plus the size indicated by `size` would be greater than the maximum physical address supported by the processor.

`MemoryInUseException`⁴⁵⁰ - Thrown if the specified memory is already in use.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

`IllegalAssignmentError`⁴⁴⁸ - Thrown if storing `logic` in this would violate the assignment rules.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException455` - Thrown if the size estimate from `size` extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException457` - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter205` has been registered with the `PhysicalMemoryManager197`.

`MemoryTypeConflictException452` - Thrown if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

```
public ImmortalPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance with the given parameters.

Parameters:

`type` - An instance of `Object` or an array of objects representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type

of memory is acceptable. Note that type values are compared by reference (==), not by value (equals).

`size` - A size estimator for this area.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`₁₆₃ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`₄₅₅ - Thrown if the `size` extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`₂₀₅ has been registered with the `PhysicalMemoryManager`₁₉₇.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`MemoryTypeConflictException`₄₅₂ - Thrown if type specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

`IllegalAssignmentError`₄₄₈ - Thrown if storing `logic` in this would violate the assignment rules.

7.11 LTPhysicalMemory

Declaration

```
public class LTPhysicalMemory extends ScopedMemory172
```

Description

An instance of `LTPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same semantics as `ScopedMemory`₁₇₂ memory areas, and the same performance restrictions as `LTMemory`₁₈₇.

No provision is made for sharing object in `LPhysicalMemory` with entities outside the JVM that creates them, and, while the memory backing an instance of `LPhysicalMemory` could be shared by multiple JVMs, the class does not support such sharing.

Methods from `LPhysicalMemory` should be overridden only by methods that use `super`.

See Also: [MemoryArea₁₆₁](#), [ScopedMemory₁₇₂](#), [VtMemory₁₉₃](#), [LTMemory₁₈₇](#), [VTPhysicalMemory₂₃₀](#), [ImmortalPhysicalMemory₂₁₂](#), [RealtimeThread₂₉](#), [NoHeapRealtimeThread₅₅](#)

7.11.1 Constructors

```
public LPhysicalMemory(
    java.lang.Object type,
    long size)
```

Create an instance of `LPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the implementation detects `size` extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException](#)₄₅₂ - Thrown if type specifies incompatible memory attributes.

See Also: [PhysicalMemoryManager](#)₁₉₇

```
public LTPhysicalMemory(
    java.lang.Object type,
    long base,
    long size)
```

Create an instance of LTPhysicalMemory with the given parameters.

Parameters:

type - An instance of Object representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, type may be an array of objects. If type is null or a reference to an array with no entries, any type of memory is acceptable. Note that type values are compared by reference (==), not by value (equals).

base - The physical memory address of the area.

size - The size of the area in bytes.

Throws:

[java.lang.SecurityException](#) - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the size is less than zero, or the implementation detects that base plus size extends beyond physically addressable memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the address is invalid.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇.

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

[java.lang.IllegalArgumentException](#) - Thrown if size is less than zero.

[MemoryInUseException](#)₄₅₀ - Thrown if the specified memory is already in use.

See Also: [PhysicalMemoryManager](#)₁₉₇

```
public LPhysicalMemory(
    java.lang.Object type,
    long base,
    long size,
    java.lang.Runnable logic)
```

Create an instance of `LPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)](#)₁₆₃ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the implementation detects that `base` plus `size` extends beyond physically addressable memory.

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the address is invalid.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no

matching `PhysicalMemoryTypeFilter205` has been registered with the `PhysicalMemoryManager197`.

`MemoryTypeConflictException452` - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

`MemoryInUseException450` - Thrown if the specified memory is already in use.

`IllegalAssignmentError448` - Thrown if storing logic in this would violate the assignment rules.

See Also: `PhysicalMemoryManager197`

```
public LTPhysicalMemory(
    java.lang.Object type,
    long size,
    java.lang.Runnable logic)
```

Create an instance of `LTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()163` is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

`java.lang.IllegalArgumentException` - Thrown if `size` is less than zero.

`SizeOutOfBoundsException455` - Thrown if the implementation detects that `size` extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇.

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

See Also: [PhysicalMemoryManager](#)₁₉₇

```
public LTPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size)
```

Create an instance of `LTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the implementation detects that base plus the size estimate extends beyond physically addressable memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the address is invalid.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no

matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

[MemoryInUseException₄₅₀](#) - Thrown if the specified memory is already in use.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public LTPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance of `LTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)₁₆₃](#) is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsExcpetion₄₅₅](#) - Thrown if the implementation detects that `base` plus the size estimate extends beyond physically addressable memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the address is invalid.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇.

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

[MemoryInUseException](#)₄₅₀ - Thrown if the specified memory is already in use.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

See Also: [PhysicalMemoryManager](#)₁₉₇

```
public LTPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size)
```

Create an instance of `LTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the implementation detects that `size` extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#) .

[MemoryTypeConflictException₄₅₂](#) - Thrown if type specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public LTPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance of `LTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - A size estimator for this area.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)₁₆₃](#) is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given type of memory.

[SizeOutOfBoundsExcep₄₅₅](#) - Thrown if the implementation detects that base plus the size estimate extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#) .

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

[IllegalAssignmentError](#)₄₄₈ - Thrown if storing logic in this would violate the assignment rules.

See Also: [PhysicalMemoryManager](#)₁₉₇

7.11.2 Methods

```
public java.lang.String toString()
```

Creates a string describing this object. The string is of the form

(LTPhysicalMemory) Scoped memory # num

where num is a number that uniquely identifies this LTPhysicalMemory memory area. representing the value of this.

Overrides: [toString](#)₁₈₇ in class [ScopedMemory](#)₁₇₂

Returns: A string representing the value of this.

7.12 VTPhysicalMemory

Declaration

```
public class VTPhysicalMemory extends ScopedMemory172
```

Description

An instance of VTPhysicalMemory allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same semantics as [ScopedMemory](#)₁₇₂ memory areas, and the same performance restrictions as VMemory.

No provision is made for sharing object in VTPhysicalMemory with entities outside the JVM that creates them, and, while the memory backing an instance of VTPhysicalMemory could be shared by multiple JVMs, the class does not support such sharing.

Methods from VTPhysicalMemory should be overridden only by methods that use `super`.

See Also: [MemoryArea₁₆₁](#), [ScopedMemory₁₇₂](#), [VTMemory₁₉₃](#), [LTMemory₁₈₇](#),
[LTPhysicalMemory₂₂₁](#), [ImmortalPhysicalMemory₂₁₂](#), [RealtimeThread₂₉](#),
[NoHeapRealtimeThread₅₅](#)

7.12.1 Constructors

public VTPhysicalMemory(java.lang.Object type, long size)

Create an instance of VTPhysicalMemory with the given parameters.

Parameters:

type - An instance of Object representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, type may be an array of objects. If type is null or a reference to an array with no entries, any type of memory is acceptable. Note that type values are compared by reference (`==`), not by value (`equals`).

size - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the implementation detects that size extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if type specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if size is less than zero.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VPhysicalMemory(
    java.lang.Object type,
    long base,
    long size)
```

Create an instance of `VPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the implementation detects that `size` extends beyond physically addressable memory.

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the base address is invalid.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

[MemoryInUseException₄₅₀](#) - Thrown if the specified memory is already in use.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VTPhysicalMemory(
    java.lang.Object type,
    long base,
    long size,
    java.lang.Runnable logic)
```

Create an instance of VTPhysicalMemory with the given parameters.

Parameters:

`type` - An instance of Object representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁶³ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`SizeOutOfBoundsException`⁴⁵⁵ - Thrown if the implementation detects that `size` extends beyond physically addressable memory.

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`⁴⁵⁴ - Thrown if the base address is invalid.

`UnsupportedPhysicalMemoryException`⁴⁵⁷ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`²⁰⁵ has been registered with the `PhysicalMemoryManager`¹⁹⁷.

`MemoryTypeConflictException`⁴⁵² - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`MemoryInUseException`⁴⁵⁰ - Thrown if the specified memory is already in use.

[IllegalAssignmentError₄₄₈](#) - Thrown if storing `logic` in this would violate the assignment rules.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VPhysicalMemory(
    java.lang.Object type,
    long size,
    java.lang.Runnable logic)
```

Create an instance of `VPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever [MemoryArea.enter\(\)₁₆₃](#) is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the implementation detects that `size` extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

[IllegalAssignmentError₄₄₈](#) - Thrown if storing `logic` in this would violate the assignment rules.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VTPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size)
```

Create an instance of VTPhysicalMemory with the given parameters.

Parameters:

type - An instance of Object representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, *type* may be an array of objects. If *type* is null or a reference to an array with no entries, any type of memory is acceptable. Note that *type* values are compared by reference (`==`), not by value (`equals`).

base - The physical memory address of the area.

size - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException455` - Thrown if the implementation detects that the size estimate from *size* extends beyond physically addressable memory.

`OffsetOutOfBoundsException454` - Thrown if the base address is invalid.

`UnsupportedPhysicalMemoryException457` - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter205` has been registered with the `PhysicalMemoryManager197`.

`MemoryTypeConflictException452` - Thrown if the specified base does not point to memory that matches the requested type, or if *type* specifies incompatible memory attributes.

`MemoryInUseException450` - Thrown if the specified memory is already in use.

`java.lang.IllegalArgumentException` - Thrown if *size* is null, or *size.getEstimate()* is negative.

See Also: `PhysicalMemoryManager197`

```
public VPhysicalMemory(
    java.lang.Object type,
    long base,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance of `VPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁶³ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`⁴⁵⁵ - Thrown if the implementation detects that the size estimate from `size` extends beyond physically addressable memory.

`OffsetOutOfBoundsException`⁴⁵⁴ - Thrown if the base address is invalid.

`UnsupportedPhysicalMemoryException`⁴⁵⁷ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`²⁰⁵ has been registered with the `PhysicalMemoryManager`¹⁹⁷.

`MemoryTypeConflictException`⁴⁵² - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`MemoryInUseException`⁴⁵⁰ - Thrown if the specified memory is already in use.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

[IllegalAssignmentError₄₄₈](#) - Thrown if storing logic in this would violate the assignment rules.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VTPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size)
```

Create an instance of `VTPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the implementation detects that the size estimate from `size` extends beyond physically addressable memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

See Also: [PhysicalMemoryManager₁₉₇](#)

```
public VPhysicalMemory(
    java.lang.Object type,
    javax.realtime.SizeEstimator170 size,
    java.lang.Runnable logic)
```

Create an instance of `VPhysicalMemory` with the given parameters.

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - A size estimator for this area.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`¹⁶³ is called. If `logic` is null, `logic` must be supplied when the memory area is entered.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`⁴⁵⁵ - Thrown if the implementation detects that the size estimate from `size` extends beyond physically addressable memory.

`UnsupportedPhysicalMemoryException`⁴⁵⁷ - Thrown if the underlying hardware does not support the given type, or if no matching `PhysicalMemoryTypeFilter`²⁰⁵ has been registered with the `PhysicalMemoryManager`¹⁹⁷.

`MemoryTypeConflictException`⁴⁵² - Thrown if the specified base does not point to memory that matches the requested type, or if `type` specifies incompatible memory attributes.

`java.lang.IllegalArgumentException` - Thrown if `size` is null, or `size.getEstimate()` is negative.

`IllegalAssignmentError`⁴⁴⁸ - Thrown if storing `logic` in this would violate the assignment rules.

See Also: `PhysicalMemoryManager`¹⁹⁷

7.12.2 Methods

```
public java.lang.String toString()
```

Creates a string representing this object. The string is of the form
(VTPhysicalMemory) Scoped memory # num

where num is a number that uniquely identifies this VTPhysicalMemory memory area.

Overrides: [toString₁₈₇](#) in class [ScopedMemory₁₇₂](#)

Returns: A string representing the value of this.

7.13 RawMemoryAccess

Declaration

```
public class RawMemoryAccess
```

Direct Known Subclasses: [RawMemoryFloatAccess₂₆₂](#)

Description

An instance of `RawMemoryAccess` models a range of physical memory as a fixed sequence of bytes. A full complement of accessor methods allow the contents of the physical area to be accessed through offsets from the base, interpreted as byte, short, int, or long data values or as arrays of these types.

Whether the offset addresses the high-order or low-order byte is normally based on the value of the [RealtimeSystem.BYTE_ORDER₄₄₂](#) static byte variable in class [RealtimeSystem₄₄₁](#). If the type of memory used for this `RawMemoryAccess` region implements non-standard byte ordering, accessor methods in this class continue to select bytes starting at `offset` from the base address and continuing toward greater addresses. The memory type may control the mapping of these bytes into the primitive data type. The memory type could even select bytes that are not contiguous. In each case the documentation for the [PhysicalMemoryTypeFilter₂₀₅](#) must document any mapping other than the “normal” one specified above.

The `RawMemoryAccess` class allows a real-time program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software.

A raw memory area cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java’s type checking) and error-

prone (since it is sensitive to the specific representational choices made by the Java compiler).

Many of the constructors and methods in this class throw `OffsetOutOfBoundsException`⁴⁵⁴. This exception means that the value given in the offset parameter is either negative or outside the memory area.

Many of the constructors and methods in this class throw `SizeOutOfBoundsException`⁴⁵⁵. This exception means that the value given in the size parameter is either negative, larger than an allowable range, or would cause an accessor method to access an address outside of the memory area.

Unlike other integral parameters in this chapter, negative values are valid for `byte`, `short`, `int`, and `long` values that are copied in and out of memory by the `set` and `get` methods of this class.

All offset values used in this class are measured in bytes.

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA), consequently atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then re-storing the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store will be lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

This class supports unaligned access to data, but it does not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulable objects. A raw memory area could be updated by another schedulable object, or even unmapped in the middle of a method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw `put` and `get`. The properties represent a four-dimensional sparse array with

boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The dimension are

Attribute	Values	Comment
Access type	read, write	
Data type	byte, short, int, long, float, double	
Alignment	0 to 7	For each data type, the possible alignments range from 0 == aligned to data size - 1 == only the first byte of the data is <i>alignment</i> bytes away from natural alignment.
Atomicity	processor, smp, memory	<i>processor</i> means access is atomic with respect to other schedulable objects on that processor. <i>smp</i> means that access is <i>processor</i> atomic, and atomic with respect across the processors in an SMP. <i>memory</i> means that access is <i>smp</i> atomic, and atomic with respect to all access to the memory including DMA.

The true values in the table are represented by properties of the following form.
javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true for
example:

```
javax.realtime.atomicaccess_read_byte_0_memory=true
```

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The run-time must be forced to re-read memory or write to memory on each call to a raw memory getxxx or putxxx method, and to complete the reads and writes in the order they appear in the program order.

7.13.1 Constructors

```
public RawMemoryAccess(
    java.lang.Object type,
    long size)
```

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. If the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `vmFlags` and `vmAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)](#)₂₀₆ and [PhysicalMemoryTypeFilter.getVMFlags\(\)](#)₂₀₆ .

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the size is negative or extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇ .

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the request type, or if `type` specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if the requested type of memory exists, but there is not enough of it free to satisfy the request.

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory or the given range of memory.

```
public RawMemoryAccess(
    java.lang.Object type,
    long base,
    long size)
```

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. If the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)206](#) and [PhysicalMemoryTypeFilter.getVMFlags\(\)206](#) .

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the region.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

[OffsetOutOfBoundsException454](#) - Thrown if the address is invalid.

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the size is negative or extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException₄₅₇](#) - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter₂₀₅](#) has been registered with the [PhysicalMemoryManager₁₉₇](#).

[MemoryTypeConflictException₄₅₂](#) - Thrown if the specified base does not point to memory that matches the request type, or if type specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if the requested type of memory exists, but there is not enough of it free to satisfy the request.

7.13.2 Methods

`public byte getBytes(long offset)`

Gets the byte at the given offset in the memory area associated with this object. The byte is always loaded from memory in a single atomic operation.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory from which to load the byte.

Returns: The byte from raw memory.

Throws:

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the byte falls in an invalid address range.

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void getBytes(  
    long offset,  
    byte[] bytes,  
    int low,  
    int number)
```

Gets number bytes starting at the given offset in the memory area associated with this object and assigns them to the byte array passed starting at position `low`. Each byte is loaded from memory in a single atomic operation. Groups of bytes may be loaded together, but this is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory from which to start loading.

`bytes` - The array into which the loaded items are placed.

`low` - The offset which is the starting point in the given array for the loaded items to be placed.

`number` - The number of items to load.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the byte falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The `bytes` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

`public int getInt(long offset)`

Gets the `int` at the given `offset` in the memory area associated with this object. If the integer is aligned on a “natural” boundary it is always loaded from memory in a single atomic operation. If it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to load the integer.

Returns: The integer from raw memory.

Throws:

`OffsetOutOfBoundsException`₄₅₄ - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`₄₅₅ somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `map(Long, Long)`₂₅₃).

`SizeOutOfBoundsException`₄₅₅ - Thrown if the object is not mapped, or if the integer falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void getInts(  
    long offset,  
    int[] ints,  
    int low,  
    int number)
```

Gets `number` integers starting at the given offset in the memory area associated with this object and assign them to the `int` array passed starting at position `low`.

If the integers are aligned on natural boundaries each integer is loaded from memory in a single atomic operation. Groups of integers may be loaded together, but this is unspecified.

If the integers are not aligned on natural boundaries they may not be loaded atomically and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start loading.

`ints` - The array into which the integers read from the raw memory are placed.

`low` - The offset which is the starting point in the given array for the loaded items to be placed.

`number` - The number of integers to loaded.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the integers fall in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The

ints array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

`public long getLong(long offset)`

Gets the `long` at the given offset in the memory area associated with this object.

The load is not required to be atomic even it is located on a natural boundary.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to load the `long`.

Returns: The `long` from raw memory.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is invalid.

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the double falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void getLongs(
    long offset,
    long[] longs,
    int low,
    int number)
```

Gets number longs starting at the given offset in the memory area associated with this object and assign them to the long array passed starting at position low.

The loads are not required to be atomic even if they are located on natural boundaries.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start loading.

`longs` - The array into which the loaded items are placed.

`low` - The offset which is the starting point in the given array for the loaded items to be placed.

`number` - The number of longs to load.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(Long, Long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if a long falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The `longs` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

`public long getMappedAddress()`

Gets the virtual memory location at which the memory region is mapped.

Returns: The virtual address to which this is mapped (for reference purposes). Same as the base address if virtual memory is not supported.

Throws:

`java.lang.IllegalStateException` - Thrown if the raw memory object is not in the mapped state.

`public short getShort(long offset)`

Gets the short at the given offset in the memory area associated with this object. If the short is aligned on a natural boundary it is always loaded from memory in a single atomic operation. If it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to load the short.

Returns: The short loaded from raw memory.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException455` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `map(Long, Long)253`).

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the short falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void getShorts(  
    long offset,  
    short[] shorts,  
    int low,  
    int number)
```

Gets number shorts starting at the given offset in the memory area associated with this object and assign them to the short array passed starting at position `low`.

If the shorts are located on natural boundaries each short is loaded from memory in a single atomic operation. Groups of shorts may be loaded together, but this is unspecified.

If the shorts are not located on natural boundaries the load may not be atomic, and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to start loading.

`shorts` - The array into which the loaded items are placed.

`low` - The offset which is the starting point in the given array for the loaded shorts to be placed.

`number` - The number of shorts to load.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility

that the memory area could be unmapped or remapped. The `shorts` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

`public long map()`

Maps the physical memory range into virtual memory. No-op if the system doesn't support virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)206](#) and [PhysicalMemoryTypeFilter.getVMFlags\(\)206](#) .

If the object is already mapped into virtual memory, this method does not change anything.

Returns: The starting point of the virtual memory range.

Throws:

`java.lang.OutOfMemoryError` - Thrown if there is insufficient free virtual address space to map the object.

`public long map(long base)`

Maps the physical memory range into virtual memory at the specified location. No-op if the system doesn't support virtual memory.

The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)206](#) and [PhysicalMemoryTypeFilter.getVMFlags\(\)206](#) .

If the object is already mapped into virtual memory at a different address, this method remaps it to `base`.

If a remap is requested while another schedulable object is accessing the raw memory, the map will block until one load or store completes. It can interrupt an array operation between entries.

Parameters:

base - The location to map at the virtual memory space.

Returns: The starting point of the virtual memory.

Throws:

`java.lang.OutOfMemoryError` - Thrown if there is insufficient free virtual memory at the specified address.

`java.lang.IllegalArgumentException` - Thrown if **base** is not a legal value for a virtual address, or the memory-mapping hardware cannot place the physical memory at the designated address.

```
public long map(long base, long size)
```

Maps the physical memory range into virtual memory. No-op if the system doesn't support virtual memory.

The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)206](#) and [PhysicalMemoryTypeFilter.getVMFlags\(\)206](#) .

If the object is already mapped into virtual memory at a different address, this method remaps it to **base**.

If a remap is requested while another schedulable object is accessing the raw memory, the map will block until one load or store completes. It can interrupt an array operation between entries.

Parameters:

base - The location to map at the virtual memory space.

size - The size of the block to map in. If the size of the raw memory area is greater than **size**, the object is unchanged but accesses beyond the mapped region will throw [SizeOutOfBoundsException455](#) . If the size of the raw memory area is smaller than the mapped region access to the raw memory will behave as if the mapped region matched the raw memory area, but additional virtual address space will be consumed after the end of the raw memory area.

Returns: The starting point of the virtual memory.

Throws:

`java.lang.IllegalArgumentException` - Thrown if size is not greater than zero, base is not a legal value for a virtual address, or the memory-mapping hardware cannot place the physical memory at the designated address.

public void setByte(long offset, byte value)

Sets the byte at the given offset in the memory area associated with this object.

This memory access may involve a load and a store, and it may have unspecified effects on surrounding bytes in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area to which to write the byte.

`value` - The byte to write.

Throws:

`OffsetOutOfBoundsException`₄₅₄ - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`₄₅₅ somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `map(long, long)`₂₅₃).

`SizeOutOfBoundsException`₄₅₅ - Thrown if the object is not mapped, or if the byte falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setBytes(  
    long offset,  
    byte[] bytes,  
    int low,  
    int number)
```

Sets `number` bytes starting at the given `offset` in the memory area associated with this object from the byte array passed starting at position `low`.

This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area to which to start writing.

`bytes` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of items to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(Long, Long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

public void setInt(long offset, int value)

Sets the `int` at the given offset in the memory area associated with this object. On most processor architectures an aligned integer can be stored in an atomic operation, but this is not required.

This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to write the integer.

`value` - The integer to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(Long, Long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the integer falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setInts(  
    long offset,  
    int[] ints,  
    int low,  
    int number)
```

Sets `number` ints starting at the given `offset` in the memory area associated with this object from the `int` array passed starting at position `low`. On most processor architectures each aligned integer can be stored in an atomic operation, but this is not required.

This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start writing.

`ints` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of items to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if an `int` falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

public void setLong(long offset, long value)

Sets the `long` at the given offset in the memory area associated with this object. Even if it is aligned, the `long` value may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to write the `long`.

`value` - The `long` to write.

Throws:

`OffsetOutOfBoundsException`₄₅₄ - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`₄₅₅ somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `map(long, long)`₂₅₃).

`SizeOutOfBoundsException`₄₅₅ - Thrown if the object is not mapped, or if the `long` falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setLongs(
    long offset,
    long[] longs,
    int low,
    int number)
```

Sets `number` longs starting at the given offset in the memory area associated with this object from the long array passed starting at position `low`. Even if they are aligned, the long values may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start writing.

`longs` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of items to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially

complete if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

public void setShort(long offset, short value)

Sets the short at the given offset in the memory area associated with this object.

This memory access may involve a load and a store, and it may have unspecified effects on surrounding shorts in the presence of concurrent access.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to write the short.

`value` - The short to write.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException455` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `map(long, long)253`).

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the short falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setShorts(
    long offset,
    short[] shorts,
    int low,
    int number)
```

Sets `number` shorts starting at the given `offset` in the memory area associated with this object from the short array passed starting at position `low`.

Each write of a short value may involve a load and a store, and it may have unspecified effects on surrounding shorts in the presence of concurrent access - even on other shorts in the array.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start writing.

`shorts` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of items to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException₄₅₅](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [map\(Long, Long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

`public void unmap()`

Unmap the physical memory range from virtual memory. This changes the raw memory from the mapped state to the unmapped state. If the platform supports virtual memory, this operation frees the virtual addresses used for the raw memory region.

If the object is already in the unmapped state, this method has no effect.

While a raw memory object is unmapped all attempts to set or get values in the raw memory will throw `SizeOutOfBoundsException`⁴⁵⁵.

An unmapped raw memory object can be returned to mapped state with any of the object's map methods.

If an unmap is requested while another schedulable object is accessing the raw memory, the unmap will throw an `IllegalStateException`. The unmap method can interrupt an array operation between entries.

7.14 RawMemoryFloatAccess

Declaration

```
public class RawMemoryFloatAccess extends RawMemoryAccess239
```

Description

This class holds the accessor methods for accessing a raw memory area by float and double types. Implementations are required to implement this class if and only if the underlying Java Virtual Machine supports floating point data types.

By default, the byte addressed by `offset` is the byte at the lowest address of the floating point processor's floating point representation. If the type of memory used for this `RawMemoryFloatAccess` region implements a non-standard floating point format, accessor methods in this class continue to select bytes starting at `offset` from the base address and continuing toward greater addresses. The memory type may control the mapping of these bytes into the primitive data type. The memory type could even select bytes that are not contiguous. In each case the documentation for the

[PhysicalMemoryTypeFilter₂₀₅](#) must document any mapping other than the “normal” one specified above.

All offset values used in this class are measured in bytes.

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA), consequently atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads for aligned floats. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor’s word size.

This class supports unaligned access to data, but it does not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to threads. A raw memory area could be updated by another thread, or even unmapped in the middle of a method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports a optional system properties that identify a platform’s level of support for atomic raw put and get. (See [RawMemoryAccess₂₃₉](#).) The properties represent a four-dimensional sparse array with boolean values whether that combination of access attributes is atomic. The default value for array entries is false.

Many of the constructors and methods in this class throw [OffsetOutOfBoundsException₄₅₄](#). This exception means that the value given in the offset parameter is either negative or outside the memory area.

Many of the constructors and methods in this class throw [SizeOutOfBoundsException₄₅₅](#). This exception means that the value given in the size parameter is either negative, larger than an allowable range, or would cause an accessor method to access an address outside of the memory area.

7.14.1 Constructors

```
public RawMemoryFloatAccess(
    java.lang.Object type,
    long size)
```

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. If the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `vmFlags` and `vmAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's type parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)](#)₂₀₆ and [PhysicalMemoryTypeFilter.getVMFlags\(\)](#)₂₀₆ .

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the size is negative or extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no matching [PhysicalMemoryTypeFilter](#)₂₀₅ has been registered with the [PhysicalMemoryManager](#)₁₉₇ .

[MemoryTypeConflictException](#)₄₅₂ - Thrown if the specified base does not point to memory that matches the request type, or if `type` specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if the requested type of memory exists, but there is not enough of it free to satisfy the request.

```
public RawMemoryFloatAccess(
    java.lang.Object type,
    long base,
    long size)
```

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. If the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the `VMFlags` and `VMAttributes` of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See [PhysicalMemoryTypeFilter.getVMAttributes\(\)](#)₂₀₆ and [PhysicalMemoryTypeFilter.getVMFlags\(\)](#)₂₀₆ .

Parameters:

`type` - An instance of `Object` representing the type of memory required (e.g., *dma*, *shared*) - used to define the base address and control the mapping. If the required memory has more than one attribute, `type` may be an array of objects. If `type` is null or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (`==`), not by value (`equals`).

`base` - The physical memory address of the region.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

[OffsetOutOfBoundsException](#)₄₅₄ - Thrown if the address is invalid.

[SizeOutOfBoundsException](#)₄₅₅ - Thrown if the size is negative or extends into an invalid range of memory.

[UnsupportedPhysicalMemoryException](#)₄₅₇ - Thrown if the underlying hardware does not support the given type, or if no

matching `PhysicalMemoryTypeFilter205` has been registered with the `PhysicalMemoryManager197`.

`MemoryTypeConflictException452` - Thrown if the specified base does not point to memory that matches the request type, or if type specifies incompatible memory attributes.

`java.lang.OutOfMemoryError` - Thrown if the requested type of memory exists, but there is not

7.14.2 Methods

public double `getDouble`(long offset)

Gets the `double` at the given offset in the memory area associated with this object.

The load is not required to be atomic even it is located on a natural boundary.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to load the long.

Returns: The double from raw memory.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is invalid.

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the double falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void getDoubles(
    long offset,
    double[] doubles,
    int low,
    int number)
```

Gets number doubles starting at the given offset in the memory area associated with this object and assign them to the double array passed starting at position low.

The loads are not required to be atomic even if they are located on natural boundaries.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start loading.

`doubles` - The array into which the loaded items are placed.

`low` - The offset which is the starting point in the given array for the loaded items to be placed.

`number` - The number of doubles to load.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [RawMemoryAccess.map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if a double falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The `doubles` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

public float getFloat(long offset)

Gets the float at the given offset in the memory area associated with this object. If the float is aligned on a “natural” boundary it is always loaded from memory in a single atomic operation. If it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area from which to load the float.

Returns: The float from raw memory.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [RawMemoryAccess.map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the float falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

**public void getFloats(
 long offset,
 float[] floats,
 int low,
 int number)**

Gets `number` floats starting at the given offset in the memory area associated with this object and assign them to the `int` array passed starting at position `low`.

If the floats are aligned on natural boundaries each float is loaded from memory in a single atomic operation. Groups of floats may be loaded together, but this is unspecified.

If the floats are not aligned on natural boundaries they may not be loaded atomically and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start loading.

`floats` - The array into which the floats loaded from the raw memory are placed.

`low` - The offset which is the starting point in the given array for the loaded items to be placed.

`number` - The number of floats to loaded.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [RawMemoryAccess.map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if a float falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The `floats` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setDouble(long offset, double value)
```

Sets the `double` at the given offset in the memory area associated with this object. Even if it is aligned, the double value may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to write the double.

`value` - The double to write.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long, long)253`).

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the double falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setDoubles(
    long offset,
    double[] doubles,
    int low,
    int number)
```

Sets `number` doubles starting at the given offset in the memory area associated with this object from the double array passed starting at position `low`. Even if they are aligned, the double values may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not

cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start writing.

`doubles` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of items to write.

Throws:

`OffsetOutOfBoundsException454` - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long, long)253`).

`SizeOutOfBoundsException455` - Thrown if the object is not mapped, or if the a short falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The `doubles` array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

public void setFloat(long offset, float value)

Sets the `float` at the given offset in the memory area associated with this object. On most processor architectures an aligned float can be stored in an atomic operation, but this is not required.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to write the integer.

`value` - The float to write.

Throws:

`OffsetOutOfBoundsException`₄₅₄ - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long, long)`₂₅₃).

`SizeOutOfBoundsException`₄₅₅ - Thrown if the object is not mapped, or if the float falls in an invalid address range.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

```
public void setFloats(
    long offset,
    float[] floats,
    int low,
    int number)
```

Sets `number` floats starting at the given `offset` in the memory area associated with this object from the float array passed starting at position `low`. On most processor architectures each aligned float can be stored in an atomic operation, but this is not required.

Caching of the memory access is controlled by the memory type requested when the `RawMemoryAccess` instance was created. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameters:

`offset` - The offset in bytes from the beginning of the raw memory area at which to start writing.

`floats` - The array from which the items are obtained.

`low` - The offset which is the starting point in the given array for the items to be obtained.

`number` - The number of floats to write.

Throws:

[OffsetOutOfBoundsException₄₅₄](#) - Thrown if the offset is negative or greater than the size of the raw memory area. The role of the [SizeOutOfBoundsException](#) somewhat overlaps this exception since it is thrown if the offset is within the object but outside the mapped area. (See [RawMemoryAccess.map\(long, long\)₂₅₃](#)).

[SizeOutOfBoundsException₄₅₅](#) - Thrown if the object is not mapped, or if the float falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The store of the array into memory could, therefore, be only partially complete if the raw memory is unmapped or remapped mid-method.

`java.lang.ArrayIndexOutOfBoundsException` - Thrown if `low` is less than 0 or greater than `bytes.length - 1`, or if `low + number` is greater than or equal to `bytes.length`.

`java.lang.SecurityException` - Thrown if this access is not permitted by the security manager.

7.15 MemoryParameters

Declaration

```
public class MemoryParameters implements java.lang.Cloneable
```

All Implemented Interfaces: `java.lang.Cloneable`

Description

Memory parameters can be given on the constructor of [RealtimeThread₂₉](#) and [AsyncEventHandler₃₉₃](#). These can be used both for the purposes of admission control by the scheduler and for the purposes of pacing the garbage collector (if any) to satisfy all of the schedulable object memory allocation rates.

The limits in a `MemoryParameters` instance are enforced when a schedulable object creates a new object, e.g., uses the `new` operation. When a schedulable object exceeds its allocation or allocation rate limit, the error is handled as if the allocation failed because of insufficient memory. The object allocation throws an `OutOfMemoryError`.

When a reference to a `MemoryParameters` object is given as a parameter to a constructor, the `MemoryParameters` object becomes bound to the object being created. Changes to the values in the `MemoryParameters` object affect the constructed object. If given to more than one constructor, then changes to the values in the `MemoryParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

A `MemoryParameters` object may be shared, but that does not cause the memory budgets reflected by the parameter to be shared among the schedulable objects that are associated with the parameter object.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

7.15.1 Fields

```
public static final long NO_MAX
```

Specifies no maximum limit.

7.15.2 Constructors

```
public MemoryParameters(
    long maxMemoryArea,
    long maxImmortal)
```

Create a `MemoryParameters` object with the given values.

Parameters:

`maxMemoryArea` - A limit on the amount of memory the schedulable object may allocate in its initial memory area. Units are in bytes.

If zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.

`maxImmortal` - A limit on the amount of memory the schedulable object may allocate in the immortal area. Units are in bytes. If

zero, no allocation allowed in immortal. To specify no limit, use `NO_MAX`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `maxMemoryArea` or `maxImmortal`.

```
public MemoryParameters(
    long maxMemoryArea,
    long maxImmortal,
    long allocationRate)
```

Create a MemoryParameters object with the given values.

Parameters:

`maxMemoryArea` - A limit on the amount of memory the schedulable object may allocate in its initial memory area. Units are in bytes. If zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.

`maxImmortal` - A limit on the amount of memory the schedulable object may allocate in the immortal area. Units are in bytes. If zero, no allocation allowed in immortal. To specify no limit, use `NO_MAX`.

`allocationRate` - A limit on the rate of allocation in the heap. Units are in bytes per second of wall clock time. If `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX`. Measurement starts when the schedulable object is first released for execution (not when it is constructed.) Enforcement of the allocation rate is an implementation option. If the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `maxMemoryArea` or `maxImmortal`, or `allocationRate`.

7.15.3 Methods

```
public java.lang.Object clone()
```

Return a clone of `this`. This method should behave effectively as if it constructed a new object with the visible values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.

Overrides: `clone` in class `Object`

Since: 1.0.1

public long getAllocationRate()

Gets the limit on the rate of allocation in the heap. Units are in bytes per second.

Returns: The allocation rate in bytes per second. If zero, no allocation is allowed in the heap. If the returned value is `NO_MAX274` then the allocation rate on the heap is uncontrolled.

public long getMaxImmortal()

Gets the limit on the amount of memory the schedulable object may allocate in the immortal area. Units are in bytes.

Returns: The limit on immortal memory allocation. If zero, no allocation is allowed in immortal memory. If the returned value is `NO_MAX274` then there is no limit for allocation in immortal memory.

public long getMaxMemoryArea()

Gets the limit on the amount of memory the schedulable object may allocate in its initial memory area. Units are in bytes.

Returns: The allocation limit in the schedulable object's initial memory area. If zero, no allocation is allowed in the initial memory area. If the returned value is `NO_MAX274` then there is no limit for allocation in the initial memory area.

public void setAllocationRate(long allocationRate)

Sets the limit on the rate of allocation in the heap.

Parameters:

`allocationRate` - Units are in bytes per second of wall-clock time. If `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX`. Measurement starts when the schedulable object starts (not when it is constructed.) Enforcement of the allocation rate is an implementation option. If the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `allocationRate`.

```
public boolean setAllocationRateIfFeasible(
    long allocationRate)
```

Sets the limit on the rate of allocation in the heap. If this `MemoryParameters` object is currently associated with one or more schedulable objects that have been passed admission control, this change in allocation rate will be submitted to admission control. The scheduler (in conjunction with the garbage collector) will either admit all the effected threads with the new allocation rate, or leave the allocation rate unchanged and cause `setAllocationRateIfFeasible` to return `false`.

Parameters:

`allocationRate` - Units are in bytes per second of wall-clock time. If `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX274`. Enforcement of the allocation rate is an implementation option. If the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Returns: True if the request was fulfilled.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `allocationRate`.

```
public boolean setMaxImmortalIfFeasible(
    long maximum)
```

Sets the limit on the amount of memory the schedulable object may allocate in the immortal area.

Parameters:

`maximum` - Units are in bytes. If zero, no allocation allowed in immortal. To specify no limit, use `NO_MAX`.

Returns: True if the value is set. False if any of the schedulable objects have already allocated more than the given value. In this case the call has no effect.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `maximum`.

```
public boolean setMaxMemoryAreaIfFeasible(
    long maximum)
```

Sets the limit on the amount of memory the schedulable object may allocate in its initial memory area.

Parameters:

`maximum` - Units are in bytes. If zero, no allocation allowed in the initial memory area. To specify no limit, use `NO_MAX`.

Returns: True if the value is set. False if any of the schedulable objects have already allocated more than the given value. In this case the call has no effect.

Throws:

`java.lang.IllegalArgumentException` - Thrown if any value other than positive, zero, or `NO_MAX` is passed as the value of `maximum`.

7.16 GarbageCollector

Declaration

```
public abstract class GarbageCollector
```

Description

The system shall provide dynamic and static information characterizing the temporal behavior and imposed overhead of any garbage collection algorithm provided by the system. This information shall be made available to applications via methods on subclasses of `GarbageCollector`. Implementations are allowed to provide any set of methods in subclasses as long as the temporal behavior and overhead are sufficiently categorized. The implementations are also required to fully document the subclasses.

A reference to the garbage collector responsible for heap memory is available from `RealtimeSystem.currentGC()`⁴⁴².

7.16.1 Constructors

```
public GarbageCollector()
```

Deprecated. 1.0.1 This class and any subclasses should be singletons.

Create an instance of `this`.

7.16.2 Methods

```
public abstract javax.realtime.RelativeTime332  
getPreemptionLatency()
```

Preemption latency is a measure of the maximum time a schedulable object may have to wait for the collector to reach a preemption-safe point.

Instances of [NoHeapRealtimeThread55](#) and async event handlers with the no-heap option preempt garbage collection immediately, but other schedulable objects must wait until the collector reaches a preemption-safe point. For many garbage collectors the only preemption safe point is at the end of garbage collection, but an implementation of the garbage collector could permit a schedulable object to preempt garbage collection before it completes. The `getPreemptionLatency` method gives such a garbage collector a way to report the worst-case interval between release of a schedulable object during garbage collection, and the time the schedulable object starts execution or gains full access to heap memory, whichever comes later.

Returns: The worst-case preemption latency of the garbage collection algorithm represented by `this`. The returned object is allocated in the current allocation context. If there is no constant that bounds garbage collector preemption latency, this method shall return a relative time with `Long.MAX_VALUE` milliseconds. The number of nanoseconds in this special value is unspecified.

Unofficial

Synchronization

This section describes classes that specifically manage synchronization. These classes:

- Allow the setting of a priority inversion control policy either as the default or for specific objects
- Allow wait-free communication between schedulable objects (especially instances of `NoHeapRealtimeThread`) and regular Java threads.

This specification strengthens the semantics of Java synchronized code by mandating monitor execution eligibility control, commonly referred to as priority inversion control. The `MonitorControl` class is defined as the superclass of all such execution eligibility control algorithms. Its subclasses `PriorityInheritance` (required) and `PriorityCeilingEmulation` (optional) avoid unbounded priority inversions, which would be unacceptable in real-time systems.

The classes in this section establish a framework for priority inversion management that applies to priority-oriented schedulers in general, and a specific set of requirements for the base priority scheduler.

The wait-free queue classes provide safe, concurrent access to data shared between instances of `NoHeapRealtimeThread` and schedulable objects subject to garbage collection delays.

Semantics and Requirements

This list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. Terminology: If an object `obj` has been assigned (either by default or via an explicit method call) the `MonitorControlPolicy` `mcp`, then `obj` is said to be *governed by* `mcp`.
2. The initial default monitor control policy shall be `PriorityInheritance`. The default policy can be altered by using the `setMonitorControl()` method.
3. Notwithstanding the preceding rule, an RTSJ implementation may allow the program to establish a different initial default monitor control policy at JVM startup. The program can query the initial default monitor control policy via the method `RealtimeSystem.getInitialMonitorControl`.
4. The `PriorityCeilingEmulation` monitor control policy is optional, since it is not widely supported by current RTOSes.
5. An implementation that provides any additional `MonitorControl` subclasses must document their effects, particularly with respect to priority inversion control.
6. An object's monitor control policy affects *any* entity that attempts to lock the object; i.e., regular Java threads as well as schedulable objects.
7. When a thread or schedulable object enters synchronized code, the target object's monitor control policy must be supported by the thread or schedulable object's scheduler; otherwise an `IllegalThreadStateException` is thrown. An implementation that defines a new `MonitorControl` subclass must document which (if any) schedulers do not support this policy.

Semantics for the Base Priority Scheduler

The following list defines the main terms and establishes the general semantics and requirements that apply to threads and schedulable objects managed by the base priority scheduler when they synchronize on objects governed by monitor control policies defined in this section.

1. Each thread or schedulable object has a *base priority* and an *active priority*. A thread or schedulable object that holds a lock on a PCE-governed object also has a *ceiling priority*.
2. The *base priority* for a thread or schedulable object `t` is initially the priority that `t` has when it is created. The base priority is updated (immediately) as an effect of

invoking any of the following methods:

- `pparams.setPriority(prio)`
if `t` is a schedulable object with `pparams` as its `SchedulingParameters`, where `pparams` is an instance of `PriorityParameters`; the new base priority is `prio`
 - `t.setSchedulingParameters(pparams)`
if `t` is a schedulable object and `pparams` is an instance of `PriorityParameters`; the new base priority is `pparams.getPriority()`
 - `t.setPriority(prio)`
if `t` is a schedulable object, the new base priority is `prio`. If it is a Java thread, the new base priority is the lesser of `prio`, and the maximum priority for `t`'s thread group.
3. When `t` does not hold any locks, its active priority is the same as its base priority. In such a situation modification of the priority of `t` through an invocation of any of the above priority-setting methods for `t` causes `t` to be placed at the tail of its relevant queue (ready, blocked on a particular object, etc.) at its new priority.
 4. When `t` holds one or more locks, then `t` has a set of *priority sources*. The *active priority* for `t` at any point in time is the maximum of the priorities associated with all of these sources. The priority sources resulting from the monitor control policies defined in this section, and their associated priorities for a schedulable object `t`, are as follows:
 - a. *Source*: `t` itself
Associated priority: The base priority for `t`
Note: This may have been changed (either synchronously or asynchronously) while `t` has been holding its lock(s).
 - b. *Source*: Each object locked by `t` and governed by a `PriorityCeilingEmulation` policy
Associated priority: The maximum value `ceil` such that `ceil` is the ceiling for a `PriorityCeilingEmulation` policy governing an object locked by `t`. This value is also referred to as the *ceiling priority* for `t`.
 - c. *Source*: Each thread or schedulable object that is attempting to synchronize on an object locked by `t` and governed by a `PriorityInheritance` policy
Associated priority: The maximum active priority over all such threads and schedulable objects
Note: This rule accounts for recursive priority inheritance.
 - d. *Source*: Each thread or schedulable object that is attempting to synchronize on an object locked by `t` and governed by a `PriorityCeilingEmulation` policy.

Associated priority: The maximum active priority over all such threads and schedulable objects

Note: This rule, which in effect allows a `PriorityCeilingEmulation` lock to behave like a `PriorityInheritance` lock, helps avoid unbounded priority inversions that could otherwise occur in the presence of nested synchronizations involving a mix of `PriorityCeilingEmulation` and `PriorityInheritance` policies.

5. The addition of a priority source for τ either leaves τ 's active priority unchanged, or increases it. If τ 's active priority is unchanged, then τ 's status in its relevant queue (e.g. blocked waiting for some object) is not affected. If τ 's active priority is increased, then τ is placed at the tail of the relevant queue at its new active priority level.
6. The removal of a priority source for τ either leaves τ 's active priority unchanged, or decreases it. If τ 's active priority is unchanged, then τ 's status in its relevant queue (e.g. blocked waiting for some object) is not affected. If τ 's active priority is decreased and τ is either ready or running, then τ must be placed at the head of the ready queue at its new active priority level, if the implementation is supporting `PriorityCeilingEmulation`. If the implementation is not supporting `PriorityCeilingEmulation` then τ should be placed at the head of the ready queue at its new active priority (Note the “should”: this behavior is optional.) If `PriorityCeilingEmulation` is not supported, the implementation must document the queue placement effect. If τ 's active priority is decreased and τ is blocked, then τ is placed in the corresponding queue at its new active priority level. Its position in the queue is implementation defined, but placement at the tail is recommended.

The above rules have the following consequences:

- A thread or schedulable object τ 's priority sources from 4.b are added and removed synchronously; i.e., they are established based on τ 's entering or leaving synchronized code. However, priority sources from 4.a, 4.c and 4.d may be added and removed asynchronously, as an effect of actions by other threads or schedulable objects.
- If a thread or schedulable object holds only one lock then, when it releases this lock, its active priority is set to its base priority.
- A thread or schedulable object's active priority is never less than its base priority.
- When a thread or schedulable object blocks at a call of `obj.wait()` it releases the lock on `obj` and hence relinquishes the priority source(s) based on `obj`'s monitor control policy. The thread or schedulable object will be queued at a new active priority that reflects the loss of these priority sources.

Since base priorities may be shared (i.e., the same `PriorityParameters` object may be associated with multiple schedulable objects), a given base priority may be the active priority for some but not all of its associated schedulable objects.

It is a consequence of other rules that, when a thread or schedulable object `t` attempts to synchronize on an object `obj` governed by a `PriorityCeilingEmulation` policy with ceiling `ceil`, then `t`'s active priority may exceed `ceil` but `t`'s base priority must not. In contrast, once `t` has successfully synchronized on `obj` then `t`'s base priority may also exceed `obj`'s monitor control policy's ceiling. Note that `t`'s base priority and/or `obj`'s monitor control policy may have been dynamically modified.

Requirements For Additional Schedulers

The following list establishes the semantics and requirements that apply to threads or schedulable objects managed by a scheduler other than the base priority scheduler when they synchronize on objects with monitor control policies defined in this section.

1. An implementation that defines a new `Scheduler` subclass must document which (if any) monitor control policies the new scheduler does not support.
2. An implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default `PriorityInheritance` instance. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the default priority scheduler found in the previous section.

Rationale

Java's rules for synchronized code provide a means for mutual exclusion but do not prevent unbounded priority inversions and thus are insufficient for real-time applications. This specification strengthens the semantics for synchronized code by mandating priority inversion control, in particular by furnishing classes for priority inheritance and priority ceiling emulation. Priority inheritance is more widely implemented in real-time operating systems and thus is required and is the initial default mechanism in this specification.

Since the same object may be accessed from synchronized code by both a `NoHeapRealtimeThread` and an arbitrary thread or schedulable object, unwanted dependencies may result. To avoid this problem, this specification provides three wait-free queue classes as an alternative means for safe, concurrent data accesses without priority inversion.

8.1 MonitorControl

Declaration

```
public abstract class MonitorControl
```

Direct Known Subclasses: [PriorityCeilingEmulation₂₈₈](#),
[PriorityInheritance₂₉₁](#)

Description

Abstract superclass for all monitor control policy objects.

8.1.1 Constructors

```
protected MonitorControl()
```

Invoked from subclass constructors.

8.1.2 Methods

```
public static javax.realtime.MonitorControl286  
getMonitorControl()
```

Gets the current default monitor control policy.

Returns: The default monitor control policy object.

```
public static javax.realtime.MonitorControl286  
getMonitorControl(java.lang.Object obj)
```

Gets the monitor control policy of the given instance of `java.lang.Object`.

Parameters:

obj - The object being queried.

Returns: The monitor control policy of the obj parameter.

Throws:

`java.lang.IllegalArgumentException` - Thrown if obj is null.

```
public static javax.realtime.MonitorControl286
    setMonitorControl(
        javax.realtime.MonitorControl286 policy)
```

Sets the *default monitor control policy*. This policy does not affect the monitor control policy of any already created object, it will, however, govern any object subsequently constructed, until either:

1. a new “per-object” policy is set for that object. This will alter the monitor control policy for a single object without changing the default policy.
2. a new default policy is set.

Like the per-object method (see [setMonitorControl\(Object, MonitorControl\)](#)₂₈₇, the setting of the default monitor control policy occurs immediately.

Parameters:

`policy` - The new monitor control policy. If null nothing happens.

Returns: The default `MonitorControl` policy that was replaced.

Throws:

`java.lang.SecurityException` - Thrown if the caller is not permitted to alter the default monitor control policy.

`java.lang.IllegalArgumentException` - Thrown if `policy` is not in immortal memory.

`java.lang.UnsupportedOperationException` - Thrown if `policy` is not a supported monitor control policy.

Since: 1.0.1 The return type is changed from `void` to `MonitorControl`.

```
public static javax.realtime.MonitorControl286
    setMonitorControl(
        java.lang.Object obj,
        javax.realtime.MonitorControl286 policy)
```

Immediately sets `policy` as the monitor control policy for `obj`.

A thread or schedulable object that is queued for the lock associated with `obj`, or is in `obj`'s wait set, is not rechecked (e.g., for a `CeilingViolationException`) under `policy`, either as part of the execution of `setMonitorControl` or when it is awakened to (re)acquire the lock.

The thread or schedulable object invoking `setMonitorControl` must already hold the lock on `obj`.

Parameters:

`obj` - The object that will be governed by the new policy.

`policy` - The new policy for the object. If null nothing will happen.

Returns: The current `MonitorControl` policy for `obj`, which will be replaced.

Throws:

`java.lang.IllegalArgumentException` - Thrown when `obj` is null or `policy` is not in immortal memory.

`java.lang.UnsupportedOperationException` - Thrown if `policy` is not a supported monitor control policy.

`java.lang.IllegalMonitorStateException` - Thrown if the caller does not hold a lock on `obj`.

Since: 1.0.1 The return type is changed from `void` to `MonitorControl`.

8.2 PriorityCeilingEmulation

Declaration

```
public class PriorityCeilingEmulation extends MonitorControl286
```

Description

Monitor control class specifying the use of the priority ceiling emulation protocol (also known as the “highest lockers” protocol). Each `PriorityCeilingEmulation` instance is immutable; it has an associated *ceiling*, initialized at construction and queryable but not updatable thereafter.

If a thread or schedulable object synchronizes on a target object governed by a `PriorityCeilingEmulation` policy, then the target object becomes a priority source for the thread or schedulable object. When the object is unlocked, it ceases serving as a priority source for the thread or schedulable object. The practical effect of this rule is that the thread or schedulable object’s active priority is boosted to the policy’s ceiling when the object is locked, and is reset when the object is unlocked. The value that it is reset to may or may not be the same as the active priority it held when the object was locked; this depends on other factors (e.g. whether the thread or schedulable object’s base priority was changed in the interim).

The implementation must perform the following checks when a thread or schedulable object `t` attempts to synchronize on a target object governed by a `PriorityCeilingEmulation` policy with ceiling `ceil`:

- `t`’s base priority does not exceed `ceil`

- t 's ceiling priority (if t is holding any other `PriorityCeilingEmulation` locks) does not exceed `ceil`.

Thus for any object `targetObj` that will be governed by priority ceiling emulation, the programmer needs to provide (via `MonitorControl.setMonitorControl(Object, MonitorControl)287`) a `PriorityCeilingEmulation` policy whose ceiling is at least as high as the maximum of the following values:

- the highest base priority of any thread or schedulable object that could synchronize on `targetObj`
- the maximum ceiling priority value that any thread or schedulable object could have when it attempts to synchronize on `targetObj`.

More formally:

- If a thread or schedulable object t whose base priority is p_1 attempts to synchronize on an object governed by a `PriorityCeilingEmulation` policy with ceiling p_2 , where $p_1 > p_2$, then a `CeilingViolationException` is thrown in t . A `CeilingViolationException` is likewise thrown in t if t is holding a `PriorityCeilingEmulation` lock and has a ceiling priority exceeding p_2 .

The values of p_1 and p_2 are passed to the constructor for the exception and may be queried by an exception handler.

A consequence of the above rule is that a thread or schedulable object may nest synchronizations on `PriorityCeilingEmulation`-governed objects as long as the ceiling for the inner lock is not less than the ceiling for the outer lock.

The possibility of nested synchronizations on objects governed by a mix of `PriorityInheritance` and `PriorityCeilingEmulation` policies requires one other piece of behavior in order to avoid unbounded priority inversions. If a thread or schedulable object holds a `PriorityInheritance` lock, then any `PriorityCeilingEmulation` lock that it either holds or attempts to acquire will exhibit priority inheritance characteristics. This rule is captured above in the definition of priority sources (4.d).

When a thread or schedulable object t attempts to synchronize on a `PriorityCeilingEmulation`-governed object with ceiling `ceil`, then `ceil` must be within the priority range allowed by t 's scheduler; otherwise, an `IllegalThreadStateException` is thrown. Note that this does not prevent a regular Java thread from synchronizing on an object governed by a `PriorityCeilingEmulation` policy with a ceiling higher than 10.

The priority ceiling for an object `obj` can be modified by invoking `MonitorControl.setMonitorControl(obj, newPCE)` where `newPCE`'s ceiling has the desired value.

See also [MonitorControl](#)₂₈₆, [PriorityInheritance](#)₂₉₁, and [CeilingViolationException](#)₄₄₇.

8.2.1 Methods

```
public int getCeiling()
```

Gets the priority ceiling for this `PriorityCeilingEmulation` object.

Returns: The priority ceiling.

Since: 1.0.1

```
public int getDefaultCeiling()
```

Deprecated. Since 1.0.1. The method name is misleading. Replaced with `getCeiling()`

Gets the priority ceiling for this `PriorityCeilingEmulation` object.

Returns: The priority ceiling.

```
public static javax.realtime.PriorityCeilingEmulation288  
getMaxCeiling()
```

Gets a `PriorityCeilingEmulation` object whose ceiling is `PriorityScheduler.instance().getMaxPriority()`. This method returns a reference to a `PriorityCeilingEmulation` object allocated in immortal memory. All invocations of this method return a reference to the same object.

Returns: A `PriorityCeilingEmulation` object whose ceiling is `PriorityScheduler.instance().getMaxPriority()`.

Since: 1.0.1

```
public static javax.realtime.PriorityCeilingEmulation288  
instance(int ceiling)
```

Return a `PriorityCeilingEmulation` object with the specified ceiling. This object is in `ImmortalMemory`. All invocations with the same ceiling value return a reference to the same object.

Parameters:

`ceiling` - Priority ceiling value.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `ceiling` is outside of the range of permitted priority values (e.g., less than `PriorityScheduler.instance().getMinPriority()` or greater than `PriorityScheduler.instance().getMaxPriority()` for the base scheduler).

Since: 1.0.1

8.3 PriorityInheritance

Declaration

```
public class PriorityInheritance extends MonitorControl286
```

Description

Singleton class specifying use of the priority inheritance protocol. If a thread or schedulable object `t1` attempts to enter code that is synchronized on an object `obj` governed by this protocol, and `obj` is currently locked by a lower-priority thread or schedulable object `t2`, then

1. If `t1`'s active priority does not exceed the maximum priority allowed by `t2`'s scheduler, then `t1` becomes a priority source for `t2`; `t1` ceases to serve as a priority source for `t2` when either `t2` releases the lock on `obj`, or `t1` ceases attempting to synchronize on `obj` (e.g., when `t1` incurs an ATC).
2. Otherwise (i.e., `t1`'s active priority exceeds the maximum priority allowed by `t2`'s scheduler), an `IllegalThreadStateException` is thrown in `t1`.

Note on the 2nd rule: throwing the exception in `t1`, rather than in `t2`, ensures that the exception is synchronous.

See also [MonitorControl₂₈₆](#) and [PriorityCeilingEmulation₂₈₈](#)

8.3.1 Methods

```
public static javax.realtime.PriorityInheritance291
    instance()
```

Return a reference to the singleton `PriorityInheritance`.

This is the default `MonitorControl` policy in effect at system startup.

The `PriorityInheritance` instance shall be allocated in `ImmortalMemory`.

8.4 WaitFreeWriteQueue

Declaration

```
public class WaitFreeWriteQueue
```

Description

A queue that can be non-blocking for producers. The `WaitFreeWriteQueue` class is intended for single-writer multiple-reader communication, although it may also be used (with care) for multiple writers. A *writer* is generally an instance of `NoHeapRealtimeThread55`, and the *readers* are generally regular Java threads or heap-using real-time threads or schedulable objects. Communication is through a bounded buffer of Objects that is managed first-in-first-out. The principal methods for this class are `write` and `read`

- The `write` method appends a new element onto the queue. It is not synchronized, and does not block when the queue is full (it returns `false` instead). Multiple writer threads or schedulable objects are permitted, but if two or more threads intend to write to the same `WaitFreeWriteQueue` they will need to arrange explicit synchronization.
- The `read` method removes the oldest element from the queue. It is synchronized, and will block when the queue is empty. It may be called by more than one reader, in which case the different callers will read different elements from the queue.

`WaitFreeWriteQueue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted from the `throws` clause. These are:

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

Including these exceptions on the `throws` clause was an error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the `catch` clause around the constructor invocation.

8.4.1 Constructors

```
public WaitFreeWriteQueue(int maximum)
```

Constructs a queue containing up to `maximum` elements in immortal memory. The queue has an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

Parameters:

`maximum` - The maximum number of elements in the queue.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `maximum` argument is less than or equal to zero.

Since: 1.0.1

```
public WaitFreeWriteQueue(
    int maximum,
    javax.realtime.MemoryArea161 memory)
```

Constructs a queue containing up to `maximum` elements in memory. The queue has an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

Note: that the wait free queue's internal queue is allocated in memory, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Parameters:

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea161` in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `maximum` argument is less than or equal to zero, or `memory` is null.

`InaccessibleAreaException449` - Thrown if `memory` is a scoped memory that is not on the caller's scope stack.

Since: 1.0.1

```
public WaitFreeWriteQueue(
    java.lang.Runnable writer,
    java.lang.Runnable reader,
    int maximum,
    javax.realtime.MemoryArea161 memory)
```

Constructs a queue in memory with an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

The `writer` and `reader` parameters, if non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (if non-null.) If `memory` is null and both `Runnables` are non-null, the constructor will select the nearest common scoped parent memory area, or if there is no such scope it will use immortal memory. If all three parameters are null, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulable objects that will access the queues; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in memory, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Parameters:

`writer` - An instance of `java.lang.Thread`, a schedulable object, or null.

`reader` - An instance of `java.lang.Thread`, a schedulable object, or null.

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea161` in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - Thrown if an argument holds an invalid value. The `writer` argument must be null, a reference to a `Thread`, or a reference to a schedulable object (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be null, a reference to a `Thread`, or a reference to a schedulable object. The `maximum` argument must be greater than zero.

`MemoryScopeException451` - Thrown if either `reader` or `writer` is non-null and the `memory` argument is not compatible with

reader and writer with respect to the assignment and access rules for memory areas.

[InaccessibleAreaException₄₄₉](#) - Thrown if memory is a scoped memory that is not on the caller's scope stack.

8.4.2 Methods

```
public void clear()
```

Sets `this` to empty.

```
public boolean force(java.lang.Object object)
```

Unconditionally insert `object` into `this`, either in a vacant position or else overwriting the most recently inserted element. The `boolean` result reflects whether, at the time that `force()` returns, the position at which `object` was inserted was vacant (`false`) or occupied (`true`).

Parameters:

`object` - A non-null `java.lang.Object` to insert.

Returns: `true` if `object` has overwritten an element that was occupied when the function returns; `false` otherwise (it has been inserted into a position that was vacant when the function returns)

Throws:

[MemoryScopeException₄₅₁](#) - Thrown if a memory access error or illegal assignment error would occur while storing `object` in the queue.

`java.lang.IllegalArgumentException` - Thrown if `object` is null.

```
public boolean isEmpty()
```

Queries the system to determine if `this` is empty.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: `True`, if `this` is empty. `False`, if `this` is not empty.

```
public boolean isFull()
```

Queries the system to determine if `this` is full.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: True, if this is full. False, if this is not full.

```
public java.lang.Object read()
    throws InterruptedException
```

A synchronized and possibly blocking operation on the queue.

Returns: The `java.lang.Object` least recently written to the queue. If this is empty, the calling thread or schedulable objects blocks until an element is inserted; when it is resumed, `read` removes and returns the element.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()` or [AsynchronouslyInterruptedException.fire\(\)](#)⁴²⁵ during the time between calling this method and returning from it.

Since: 1.0.1 Throws `InterruptedException`

```
public int size()
```

Queries the queue to determine the number of elements in this.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: The number of positions in this occupied by elements that have been written but not yet read.

```
public boolean write(java.lang.Object object)
```

Inserts `object` into this if this is non-full and otherwise has no effect on this; the `boolean` result reflects whether `object` has been inserted. If the queue was empty and one or more threads or schedulable objects were waiting to read, then one will be awakened after the write. The choice of which to awaken depends on the involved scheduler(s).

Parameters:

`object` - A non-null `java.lang.Object` to insert.

Returns: true if the queue was non-full; false otherwise.

Throws:

`MemoryScopeException`₄₅₁ - Thrown if a memory access error or illegal assignment error would occur while storing object in the queue.

`java.lang.IllegalArgumentException` - Thrown if object is null.

8.5 WaitFreeReadQueue

Declaration

```
public class WaitFreeReadQueue
```

Description

A queue that can be non-blocking for consumers. The `WaitFreeReadQueue` class is intended for single-reader multiple-writer communication, although it may also be used (with care) for multiple readers. A *reader* is generally an instance of `NoHeapRealtimeThread`₅₅, and the *writers* are generally regular Java threads or heap-using real-time threads or schedulable objects. Communication is through a bounded buffer of Objects that is managed first-in-first-out. The principal methods for this class are `write` and `read`

- The `write` method appends a new element onto the queue. It is synchronized, and blocks when the queue is full. It may be called by more than one writer, in which case, the different callers will write to different elements of the queue.
- The `read` method removes the oldest element from the queue. It is not synchronized and does not block; it will return `null` when the queue is empty. Multiple reader threads or schedulable objects are permitted, but if two or more intend to read from the same `WaitFreeWriteQueue` they will need to arrange explicit synchronization.

For convenience, and to avoid requiring a reader to poll until the queue is non-empty, this class also supports instances that can be accessed by a reader that blocks on queue empty. To obtain this behavior, the reader needs to invoke the `waitForData()` method on a queue that has been constructed with a `notify` parameter set to `true`.

`WaitFreeReadQueue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted. These are

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

These exceptions were in error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the catch clause around the constructor invocation.

8.5.1 Constructors

```
public WaitFreeReadQueue(int maximum, boolean notify)
```

Constructs a queue containing up to `maximum` elements in immortal memory. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Parameters:

`maximum` - The maximum number of elements in the queue.

`notify` - A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `maximum` argument is less than or equal to zero.

Since: 1.0.1

```
public WaitFreeReadQueue(
    int maximum,
    javax.realtime.MemoryArea161 memory,
    boolean notify)
```

Constructs a queue containing up to `maximum` elements in memory. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Note: that the wait free queue's internal queue is allocated in memory, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Parameters:

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea161` in which this object and internal elements are allocated.

`notify` - A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the maximum argument is less than or equal to zero, or memory is null.

`InaccessibleAreaException449` - Thrown if memory is a scoped memory that is not on the caller's scope stack.

Since: 1.0.1

```
public WaitFreeReadQueue(
    java.lang.Runnable writer,
    java.lang.Runnable reader,
    int maximum,
    javax.realtime.MemoryArea161 memory)
```

Constructs a queue containing up to maximum elements in memory. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

The `writer` and `reader` parameters, if non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (if non-null.) If `memory` is null and both `Runnables` are non-null, the constructor will select the nearest common scoped parent memory area, or if there is no such scope it will use immortal memory. If all three parameters are null, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulable objects that will access the queue; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in memory, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Parameters:

`writer` - An instance of `java.lang.Runnable` or null.

`reader` - An instance of `java.lang.Runnable` or null.

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea161` in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - Thrown if an argument holds an invalid value. The `writer` argument must be null, a reference to a `Thread`, or a reference to a schedulable object (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be null, a reference to a `Thread`, or a reference to a schedulable object. The `maximum` argument must be greater than zero.

`MemoryScopeException`₄₅₁ - Thrown if either `reader` or `writer` is non-null and the `memory` argument is not compatible with `reader` and `writer` with respect to the assignment and access rules for memory areas.

`InaccessibleAreaException`₄₄₉ - Thrown if `memory` is a scoped memory that is not on the caller's scope stack.

```
public WaitFreeReadQueue(
    java.lang.Runnable writer,
    java.lang.Runnable reader,
    int maximum,
    javax.realtime.MemoryArea161 memory,
    boolean notify)
```

Constructs a queue containing up to `maximum` elements in memory. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

The `writer` and `reader` parameters, if non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (if non-null.) If `memory` is null and both `Runnables` are non-null, the constructor will select the nearest common scoped parent memory area, or if there is no such scope it will use immortal memory. If all three parameters are null, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulable objects that will access the queue; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Parameters:

`writer` - An instance of `java.lang.Runnable` or null.

`reader` - An instance of `java.lang.Runnable` or null.

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea161` in which internal elements are allocated.

`notify` - A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws:

`java.lang.IllegalArgumentException` - Thrown if an argument holds an invalid value. The `writer` argument must be null, a reference to a `Thread`, or a reference to a schedulable object (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be null, a reference to a `Thread`, or a reference to a schedulable object. The `maximum` argument must be greater than zero.

`InaccessibleAreaException449` - Thrown if `memory` is a scoped memory that is not on the caller's scope stack.

`MemoryScopeException451` - Thrown if either `reader` or `writer` is non-null and the `memory` argument is not compatible with `reader` and `writer` with respect to the assignment and access rules for memory areas.

8.5.2 Methods

```
public void clear()
```

Sets `this` to empty.

Note: This method needs to be used with care. Invoking `clear` concurrently with `read` or `write` can lead to unexpected results.

```
public boolean isEmpty()
```

Queries the queue to determine if `this` is empty.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: `true` if `this` is empty; `false` if `this` is not empty.

```
public boolean isFull()
```

Queries the system to determine if `this` is full.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: `true` if `this` is full; `false` if `this` is not full.

```
public java.lang.Object read()
```

Reads the least recently inserted element from the queue and returns it as the result, unless the queue is empty. If the queue is empty, `null` is returned.

Returns: The `java.lang.Object` read, or else `null` if `this` is empty.

```
public int size()
```

Queries the queue to determine the number of elements in `this`.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Returns: The number of positions in `this` occupied by elements that have been written but not yet read.

```
public void waitForData()
    throws InterruptedException
```

If `this` is empty block until a writer inserts an element.

Note: If there is a single reader and no asynchronous invocation of `clear`, then it is safe to invoke `read` after `waitForData` and know that `read` will find the queue non-empty.

Implementation note: To avoid reader and writer synchronizing on the same object, the reader should not be notified directly by a writer. (This is the issue that the non-wait queue classes are intended to solve).

Throws:

`java.lang.UnsupportedOperationException` - Thrown if `this` has not been constructed with `notify` set to `true`.

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`⁴²⁵ during the time between calling this method and returning from it.

Since: 1.0.1 `InterruptedException` was added to the `throws` clause.

```
public void write(java.lang.Object object)
    throws InterruptedException
```

A synchronized and blocking write. This call blocks on queue full and will wait until there is space in the queue.

Parameters:

object - The `java.lang.Object` that is placed in the queue.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`⁴²⁵ during the time between calling this method and returning from it.

`MemoryScopeException`⁴⁵¹ - Thrown if a memory access error or illegal assignment error would occur while storing object in the queue.

Since: 1.0.1 The return type is changed to void since it *always* returned true, and `InterruptedException` was added to the throws clause.

8.6 WaitFreeDequeue

Declaration

```
public class WaitFreeDequeue
```

Description

Deprecated. 1.0.1

A `WaitFreeDequeue` encapsulates a `WaitFreeWriteQueue` and a `WaitFreeReadQueue`. Each method on a `WaitFreeDequeue` corresponds to an equivalent operation on the underlying `WaitFreeWriteQueue` or `WaitFreeReadQueue`.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted from the throws clause. These are:

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

Including these exceptions on the `throws` clause was an error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the catch clause around the constructor invocation.

`WaitFreeDequeue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management.

8.6.1 Constructors

```
public WaitFreeDequeue(
    java.lang.Runnable writer,
    java.lang.Runnable reader,
    int maximum,
    javax.realtime.MemoryArea161 memory)
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

Constructs a queue, in memory, with an underlying [WaitFreeWriteQueue₂₉₂](#) and [WaitFreeReadQueue₂₉₇](#), each of size `maximum`.

The `writer` and `reader` parameters, if non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (if non-null.) If `memory` is null and both `Runnable`s are non-null, the constructor will select the nearest common scoped parent memory area, or if there is no such scope it will use immortal memory. If all three parameters are null, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulable objects that will access the queue; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queues' internal queues are allocated in memory, but the memory area of the wait free dequeue instance itself is determined by the current allocation context.

Parameters:

`writer` - An instance of `java.lang.Runnable` or null.

`reader` - An instance of `java.lang.Runnable` or null.

`maximum` - Then maximum number of elements in the both the [WaitFreeReadQueue₂₉₇](#) and the [WaitFreeWriteQueue₂₉₂](#).

memory - The [MemoryArea₁₆₁](#) in which internal elements are allocated.

Throws:

[MemoryScopeException₄₅₁](#) - Thrown if either reader or writer is non-null and the memory argument is not compatible with reader and writer with respect to the assignment and access rules for memory areas.

`java.lang.IllegalArgumentException` - If an argument holds an invalid value. The writer argument must be null, a reference to a Thread, or a reference to a schedulable object (a `RealtimeThread`, or an `AsyncEventHandler`.) The reader argument must be null, a reference to a Thread, or a reference to a schedulable object. The maximum argument must be greater than zero.

[InaccessibleAreaException₄₄₉](#) - Thrown if memory is a scoped memory that is not on the caller's scope stack.

8.6.2 Methods

```
public java.lang.Object blockingRead()
    throws InterruptedException
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

A synchronized call of the `read()` method of the underlying [WaitFreeWriteQueue₂₉₂](#). This call blocks on queue empty and will wait until there is an element in the queue to return.

Returns: The `java.lang.Object` read.

Throws:

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()` or [AsynchronouslyInterruptedException.fire\(\)₄₂₅](#) during the time between calling this method and returning from it.

Since: 1.0.1 Added throws `InterruptedException`.

```
public void blockingWrite(java.lang.Object object)
    throws InterruptedException
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

A synchronized call of the `write()` method of the underlying [WaitFreeReadQueue₂₉₇](#). This call blocks on queue full and waits until there is space in this.

Parameters:

`object` - The `java.lang.Object` to place in this.

Throws:

[MemoryScopeException₄₅₁](#) - Thrown if a memory access error or illegal assignment error would occur while storing object in the queue.

`java.lang.InterruptedException` - Thrown if the thread is interrupted by `interrupt()` or [AsynchronouslyInterruptedException.fire\(\)₄₂₅](#) during the time between calling this method and returning from it.

Since: 1.0.1 Return type changed from `boolean` to `void` because this method *always* returned `true`, and added `InterruptedException`.

```
public boolean force(java.lang.Object object)
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

If this's underlying [WaitFreeWriteQueue₂₉₂](#) is full, then overwrite with `object` the most recently inserted element. Otherwise this call is equivalent to `nonBlockingWrite()`.

Parameters:

`object` - The object to be written.

Returns: `true` if an element was overwritten; `false` if there as an empty element into which the write occurred.

Throws:

[MemoryScopeException₄₅₁](#) - Thrown if a memory access error or illegal assignment error would occur while storing object in the queue.

```
public java.lang.Object nonBlockingRead()
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

An unsynchronized call of the `read()` method of the underlying [WaitFreeReadQueue₂₉₇](#).

Returns: A `java.lang.Object` object read from `this`. If there are no elements in `this` then `null` is returned.

```
public boolean nonBlockingWrite(  
    java.lang.Object object)
```

Deprecated. 1.0.1 Use [WaitFreeReadQueue₂₉₇](#) and [WaitFreeWriteQueue₂₉₂](#).

An unsynchronized call of the `write()` method of the underlying [WaitFreeWriteQueue₂₉₂](#). This call does not block on queue full.

Parameters:

`object` - The `java.lang.Object` to attempt to place in `this`.

Returns: `true` if `object` was inserted (i.e., the queue was not full), `false` otherwise.

Throws:

[MemoryScopeException₄₅₁](#) - Thrown if a memory access error or illegal assignment error would occur while storing `object` in the queue.

Unofficial

Chapter 9

Time

This section contains classes that describe high-resolution time. These classes:

- Allow description of a point in time with up to nanosecond accuracy and precision (actual accuracy and precision is dependent on the precision of the underlying system).
- Allow distinctions between absolute points in time, and times relative to some starting point.

Definitions

The following terms and abbreviations will be used:

A *time object* is an instance of `AbsoluteTime`, `RelativeTime`, or `RationalTime`.

A *time object* is always associated with a `clock`. By default it is associated with the real-time clock.

The *Epoch* is the standard base time, conventionally January 1 00:00:00 GMT 1970. It is the point from which the real-time clock measures absolute time.

The *time value representation* is a compound format composed of 64 bits of millisecond timing, and 32 bits of nanoseconds within a millisecond. The millisecond constituent uses the 64 bits of a Java `long` while the nanosecond constituent uses the 32 bits of a Java `int`.

The *normalized (canonical) form* for time objects uniquely specifies the values for the millisecond and nanosecond components of a point in time, including the case

of 0 milliseconds or 0 nanoseconds, and a negative time value, according to the following three rules:

1. When both millisecond and nanosecond components are nonzero they have the same sign. The algebraic time value of the time object is the algebraic sum of the two components.
2. The millisecond component represents the algebraic number of milliseconds in the time object, with a range of $[-2^{63}, 2^{63}-1]$
3. The nanosecond component represents the algebraic number of nanoseconds within a millisecond in the time object, that is $[-10^6+1, 10^6-1]$.

Instances of `HighResolutionTime` classes always hold a normalized form of a time value. Values that cannot be normalized are not valid; for example, (`MAX_LONG` milliseconds, `MAX_INT` nanoseconds) cannot be normalized and is an illegal value.

The following table has examples of normalized representations.

time in ns	millis	nanos
2000000	2	0
1999999	1	999999
1000001	1	1
1	0	1
0	0	0
-1	0	-1
-999999	0	-999999
-1000000	-1	0
-1000001	-1	-1

Overview

The time classes required by the specification are `HighResolutionTime`, `AbsoluteTime`, `RelativeTime`, and `RationalTime` (now deprecated).

Instances of `HighResolutionTime` are not created, as the class exists to provide an implementation of the other three classes. An instance of `AbsoluteTime` encapsulates an absolute time. An instance of `RelativeTime` encapsulates a point in time that is relative to some other time value.

Instances of `RationalTime` (now deprecated) express a frequency by a numerator of type `long` (the frequency) and a denominator of type `RelativeTime`. When instances of `RationalTime` are used they shall behave like `RelativeTime` objects with a value of `interval` divided by `frequency`.

All methods returning a time object come in both allocating and non-allocating form.

Semantics and Requirements

This list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. All time objects must maintain nanosecond precision and report their values in terms of millisecond and nanosecond constituents.
2. Time objects must be constructed from other time objects, from millisecond/nanosecond values, from a `java.util.Date` or obtained as a result of invocations of methods on instances of the `Clock` class.
3. Time objects maintain and report time values in normalized form, but the normalized form is not required for input parameter values. This allows computations individually with time constituent parts using the full *signed* range and restrictions of the underlying type.
 - a. Normalization is accomplished upon method invocation by methods that accept a time object represented with individual component parts, and executed as if:
 - i. The nanosecond parameter value, which may be negative, is algebraically added to the scaled millisecond parameter value. The sign of the result provides the sign for any nonzero resulting component.
 - ii. The absolute of the result is then partitioned, giving the number of integral milliseconds for the millisecond component, while the remaining fractional part provides the number of nanoseconds for the nanosecond component.
 - iii. The resulting components are then represented, and reported when necessary, with the above computed sign.
 - b. Normalization is also performed on the result of operations by methods that perform time object addition and subtraction. Operations are executed using the appropriate arithmetic precision. If the final result of an operation can be

- represented in normalized form, then the operation must not throw arithmetic exceptions while producing intermediate results.
- c. The results of time objects operations and the normalization of results of operations performed with `millis` and `nanos`, individually as `Java long` and `Java int` types respectively, are not always equivalent. This is due to the possibility of overflow for `nanos` values outside of the normalized nanosecond range, that is $[-10^6+1, 10^6-1]$, when performing operations as `int` types, while the same values could be handled with no overflow in time object operations.
 - d. When invoking setter methods that take as a parameter only one of the two time value components, the other component has implicitly the value of 0.
4. Although logically a negative time may represent time before the Epoch or a negative time interval involved in time operations, an `Exception` may be thrown if a negative absolute time or a negative time interval is given as a parameter to methods. In general, the time values accepted by a method may be a subset of the full time values range, and depend on the method.
 5. A *time object* is always associated with a `clock`. By default it is associated with the real-time clock. Clocks are involved both in the setting as well as the usage of time objects, for example in comparisons.
 6. Methods are provided to facilitate the handling of time objects generically via the `HighResolutionTime` class. These methods allow the conversion, according to a `clock`, between `AbsoluteTime` objects and `RelativeTime` objects. These methods also allow the change of `clock` association of a time object. Note that the conversions depend on the time at which they are performed. The semantics of these operations are listed in the following table:

clock association and conversion <code>this</code> has <code>clock_a</code> and <code>ms,ns</code>	returned/updated object
<code>this_is_absolute.absolute(clock_a)</code>	<code>clock_a</code> <code>ms,ns</code>
<code>this_is_absolute.absolute(clock_b)</code>	<code>clock_b</code> <code>ms,ns</code>
<code>this_is_absolute.absolute(null)</code>	<code>real-time_clock</code> <code>ms,ns</code>
<code>this_is_absolute.relative(clock_a)</code>	<code>clock_a</code> <code>clock_a.getTime().subtract(ms,ns)</code>

<code>this_is_absolute.relative(clock_b)</code>	clock_b clock_b.getTime().subtract(ms,ns)
<code>this_is_absolute.relative(null)</code>	real-time_clock real-time_clock.getTime().subtract(ms,ns)
<code>this_is_relative.relative(clock_a)</code>	clock_a ms,ns
<code>this_is_relative.relative(clock_b)</code>	clock_b ms,ns
<code>this_is_relative.relative(null)</code>	real-time_clock ms,ns
<code>this_is_relative.absolute(clock_a)</code>	clock_a clock_a.getTime().add(ms,ns)
<code>this_is_relative.absolute(clock_b)</code>	clock_b clock_b.getTime().add(ms,ns)
<code>this_is_relative.absolute(null)</code>	real-time_clock real-time_clock.getTime().add(ms,ns)

7. Time objects must implement the `Comparable` interface if it is available. The `compareTo()` method must be implemented even if the interface is not available.

Rationale

Time is the essence of real-time systems, and a method of expressing absolute time with sub-millisecond precision is an absolute minimum requirement. Expressing time in terms of nanoseconds has precedent and allows the implementation to provide time-based services, such as timers, using whatever precision it is capable of while the application requirements are expressed to an arbitrary level of precision.

The standard Java `java.util.Date` class uses milliseconds as its basic unit in order to provide sufficient range for a wide variety of applications. Real-time programming generally requires finer resolution, and nanosecond resolution is fine enough for most purposes, but even a 64 bit real-time clock based in nanoseconds would have insufficient range in some situations, so a compound format composed of 64 bits of millisecond timing, and 32 bits of nanoseconds within a millisecond, was chosen.

The expression of millisecond and nanosecond constituents is consistent with other Java interfaces.

The expression of relative times allows for time-based metaphors such as deadline-based periodic scheduling where the cost of the task is expressed as a relative time and deadlines are usually represented as times relative to the beginning of the period.

9.1 HighResolutionTime

Declaration

```
public abstract class HighResolutionTime implements
    java.lang.Comparable, java.lang.Cloneable
```

All Implemented Interfaces: `java.lang.Cloneable`, `java.lang.Comparable`

Direct Known Subclasses: [AbsoluteTime₃₂₀](#), [RelativeTime₃₃₂](#)

Description

Class `HighResolutionTime` is the base class for `AbsoluteTime`, `RelativeTime`, `RationalTime`. Used to express time with nanosecond accuracy. This class is never used directly: it is abstract and has no public constructor. Instead, one of its subclasses [AbsoluteTime₃₂₀](#), [RelativeTime₃₃₂](#), or [RationalTime₃₄₁](#) should be used. When an API is defined that has an `HighResolutionTime` as a parameter, it can take either an absolute, relative, or rational time and will do something appropriate.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

9.1.1 Methods

```
public abstract javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock)
```

Convert the time of `this` to an absolute time, using the given instance of [Clock₃₅₂](#) to determine the current time when necessary. If `clock` is `null` the real-time clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `clock` passed as a parameter. See the derived class comments for more specific information.

Parameters:

`clock` - The instance of [Clock₃₅₂](#) used to convert the time of `this` into absolute time, and the new clock association for the result.

Returns: The `AbsoluteTime` conversion in a newly allocated object, associated with the `clock` parameter.

```
public abstract javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock,
    javax.realtime.AbsoluteTime320 dest)
```

Convert the time of `this` to an absolute time, using the given instance of `Clock352` to determine the current time when necessary. If `clock` is `null` the real-time clock is assumed. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock association of the result is with the `clock` passed as a parameter. See the derived class comments for more specific information.

Parameters:

`clock` - The instance of `Clock352` used to convert the time of `this` into absolute time, and the new clock association for the result.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The `AbsoluteTime` conversion in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

```
public java.lang.Object clone()
```

Return a clone of `this`. This method should behave effectively as if it constructed a new object with the visible values of `this`. The new object is created in the current allocation context.

Overrides: `clone` in class `Object`

Since: 1.0.1

```
public int compareTo(
    javax.realtime.HighResolutionTime314 time)
```

Compares `this` `HighResolutionTime` with the specified `HighResolutionTime` `time`.

Parameters:

`time` - Compares with the time of `this`.

Throws:

`java.lang.ClassCastException` - Thrown if the `time` parameter is not of the same class as `this`.

`java.lang.IllegalArgumentException` - Thrown if the time parameter is not associated with the same clock as `this`, or when the time parameter is `null`.

public int compareTo(`java.lang.Object` object)

For the `Comparable` interface.

Specified By: `compareTo` in interface `Comparable`

Throws:

`java.lang.IllegalArgumentException` - Thrown if the object parameter is not associated with the same clock as `this`, or when the object parameter is `null`.

`java.lang.ClassCastException` - Thrown if the specified object's type prevents it from being compared to `this` `Object`.

public boolean equals(
 `javax.realtime.HighResolutionTime`₃₁₄ time)

Returns `true` if the argument `time` has the same type and values as `this`.

Equality includes clock association.

Parameters:

time - Value compared to `this`.

Returns: `true` if the parameter `time` is of the same type and has the same values as `this`.

public boolean equals(`java.lang.Object` object)

Returns `true` if the argument `object` has the same type and values as `this`.

Equality includes clock association.

Overrides: `equals` in class `Object`

Parameters:

object - Value compared to `this`.

Returns: `true` if the parameter `object` is of the same type and has the same values as `this`.

public `javax.realtime.Clock`₃₅₂ **getClock**()

Returns a reference to the clock associated with `this`.

Returns: A reference to the `clock` associated with `this`.

Since: 1.0.1

```
public final long getMilliseconds()
```

Returns the milliseconds component of `this`.

Returns: The milliseconds component of the time represented by `this`.

```
public final int getNanoseconds()
```

Returns the nanoseconds component of `this`.

Returns: The nanoseconds component of the time represented by `this`.

```
public int hashCode()
```

Returns a hash code for this object in accordance with the general contract of `java.lang.Object.hashCode()`. Time objects that are [equal](#)₃₁₆ have the same hash code.

Overrides: `hashCode` in class `Object`

Returns: The hashcode value for this instance.

```
public abstract javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock)
```

Convert the time of `this` to a relative time, using the given instance of [Clock](#)₃₅₂ to determine the current time when necessary. If `clock` is `null` the real-time clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `clock` passed as a parameter. See the derived class comments for more specific information.

Parameters:

`clock` - The instance of [Clock](#)₃₅₂ used to convert the time of `this` into relative time, and the new clock association for the result.

Returns: The `RelativeTime` conversion in a newly allocated object, associated with the `clock` parameter.

```
public abstract javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock,
    javax.realtime.RelativeTime332 dest)
```

Convert the time of `this` to a relative time, using the given instance of [Clock](#)₃₅₂ to determine the current time when necessary. If `clock` is `null`

the real-time clock is assumed. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock association of the result is with the `clock` passed as a parameter. See the derived class comments for more specific information.

Parameters:

- `clock` - The instance of `CLock352` used to convert the time of `this` into relative time, and the new clock association for the result.
- `dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The `RelativeTime` conversion in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

```
public void set(javax.realtime.HighResolutionTime314 time)
```

Change the value represented by `this` to that of the given `time`. If the `time` parameter is `null` this method will throw `IllegalArgumentException`. If the type of `this` and the type of the given `time` are not the same this method will throw `ClassCastException`. The `clock` associated with `this` is set to be the `clock` associated with the `time` parameter.

Parameters:

- `time` - The new value for `this`.

Throws:

- `java.lang.IllegalArgumentException` - Thrown if the parameter `time` is `null`.
- `java.lang.ClassCastException` - Thrown if the type of `this` and the type of the parameter `time` are not the same.

Since: 1.0.1 The description of the method in 1.0 was erroneous.

```
public void set(long millis)
```

Sets the millisecond component of `this` to the given argument, and the nanosecond component of `this` to 0. This method is equivalent to `set(millis, 0)`.

Parameters:

- `millis` - This value shall be the value of the millisecond component of `this` at the completion of the call.

```
public void set(long millis, int nanos)
```

Sets the millisecond and nanosecond components of `this`. The setting is subject to parameter normalization. If there is an overflow in the millisecond component while normalizing then an `IllegalArgumentException` will be thrown. If after normalization the time is negative then the time represented by `this` is set to a negative value, but note that negative times are not supported everywhere. For instance, a negative relative time is an invalid value for a periodic thread's period.

Parameters:

`millis` - The desired value for the millisecond component of `this` at the completion of the call. The actual value is the result of parameter normalization.

`nanos` - The desired value for the nanosecond component of `this` at the completion of the call. The actual value is the result of parameter normalization.

Throws:

`java.lang.IllegalArgumentException` - Thrown if there is an overflow in the millisecond component while normalizing.

```
public static void waitForObject(
    java.lang.Object target,
    javax.realtime.HighResolutionTime314 time)
    throws InterruptedException
```

Behaves exactly like `target.wait()` but with the enhancement that it waits with a precision of `HighResolutionTime`. As for `target.wait()`, there is the possibility of spurious wakeup behavior.

The wait time may be relative or absolute, and it is controlled by the `clock` associated with it. If the wait time is relative, then the calling thread is blocked waiting on `target` for the amount of time given by `time`, and measured by the associated `clock`. If the wait time is absolute, then the calling thread is blocked waiting on `target` until the indicated `time` value is reached by the associated `clock`.

Parameters:

`target` - The object on which to wait. The current thread must have a lock on the object.

`time` - The time for which to wait. If it is `RelativeTime(0,0)` then wait indefinitely. If it is `null` then wait indefinitely.

Throws:

`java.lang.InterruptedException` - Thrown if this schedulable object is interrupted by `RealtimeThread.interrupt()`³⁶ or `AsynchronouslyInterruptedException.fire()`⁴²⁵ while it is waiting.

`java.lang.IllegalArgumentException` - Thrown if time represents a relative time less than zero.

`java.lang.IllegalMonitorStateException` - Thrown if target is not locked by the caller.

`java.lang.UnsupportedOperationException` - Thrown if the wait operation is not supported using the `clock` associated with time.

See Also: `java.lang.Object.wait()`, `java.lang.Object.wait(long)`, `java.lang.Object.wait(long, int)`

9.2 AbsoluteTime

Declaration

`public class AbsoluteTime` extends `HighResolutionTime`³¹⁴

All Implemented Interfaces: `java.lang.Cloneable`, `java.lang.Comparable`

Description

An object that represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by the `clock`. For the default real-time clock the fixed point is the Epoch (January 1, 1970, 00:00:00 GMT). The correctness of the Epoch as a time base depends on the real-time clock synchronization with an external world time reference. This representation was designed to be compatible with the standard Java representation of an absolute time in the `java.util.Date` class.

A time object in normalized form represents negative time if both components are nonzero and negative, or one is nonzero and negative and the other is zero. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

9.2.1 Constructors

```
public AbsoluteTime()
```

Equivalent to `new AbsoluteTime(0,0)`.

The clock association is implicitly made with the real-time clock.

```
public AbsoluteTime(javax.realtime.AbsoluteTime320 time)
```

Make a new `AbsoluteTime` object from the given `AbsoluteTime` object.

The new object will have the same clock association as the `time` parameter.

Parameters:

`time` - The `AbsoluteTime` object which is the source for the copy.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `time` parameter is null.

```
public AbsoluteTime(  
    javax.realtime.AbsoluteTime320 time,  
    javax.realtime.Clock352 clock)
```

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 the real-time clock.

Parameters:

`time` - The `AbsoluteTime` object which is the source for the copy.

`clock` - The clock providing the association for the newly constructed object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `time` parameter is null.

Since: 1.0.1

```
public AbsoluteTime(javax.realtime.Clock352 clock)
```

Equivalent to `new AbsoluteTime(0,0, clock)`.

The clock association is made with the `clock` parameter. If `clock` is null the association is made with the real-time clock.

Parameters:

`clock` - The clock providing the association for the newly constructed object.

Since: 1.0.1

```
public AbsoluteTime(java.util.Date date)
```

Equivalent to `new AbsoluteTime (date.getTime(), 0)`.

The clock association is implicitly made with the real-time clock.

Parameters:

`date` - The `java.util.Date` representation of the time past the Epoch.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the date parameter is null.

```
public AbsoluteTime(
    java.util.Date date,
    javax.realtime.Clock352 clock)
```

Equivalent to `new AbsoluteTime (date.getTime(), 0, clock)`.

Warning: While the date is used to set the milliseconds component of the new `AbsoluteTime` object (with nanoseconds component set to 0), the new object represents the date only if the `clock` parameter has an epoch equal to Epoch.

The clock association is made with the `clock` parameter. If `clock` is null the association is made with the real-time clock.

Parameters:

`date` - The `java.util.Date` representation of the time past the Epoch.

`clock` - The clock providing the association for the newly constructed object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the date parameter is null.

Since: 1.0.1

```
public AbsoluteTime(long millis, int nanos)
```

Construct an `AbsoluteTime` object with time millisecond and nanosecond components past the real-time 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 `millis` and `nanos` parameters normalization. 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 the real-time clock.

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` - Thrown if there is an overflow in the millisecond component when normalizing.

```
public AbsoluteTime(
    long millis,
    int nanos,
    javax.realtime.Clock352 clock)
```

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 `millis` and `nanos` parameters normalization. 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 the real-time clock.

Note: The start of a clock's epoch is an attribute of the clock. It is defined as the Epoch (00:00:00 GMT on Jan 1, 1970) for the default real-time clock, but other classes of clock may define other epochs.

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` - Thrown if there is an overflow in the millisecond component when normalizing.

Since: 1.0.1

9.2.2 Methods

```
public javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock)
```

Return a copy of `this` modified if necessary to have the specified clock association. A new object is allocated for the result. This method is the implementation of the abstract method of the `HighResolutionTime` base class. No conversion into `AbsoluteTime` is needed in this case. The clock association of the result is with the `clock` passed as a parameter. If `clock` is null the association is made with the real-time clock.

Overrides: `absolute314` in class `HighResolutionTime314`

Parameters:

`clock` - The `clock` parameter is used only as the new clock association with the result, since no conversion is needed.

Returns: The copy of `this` in a newly allocated `AbsoluteTime` object, associated with the `clock` parameter.

```
public javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock,
    javax.realtime.AbsoluteTime320 dest)
```

Return a copy of `this` modified if necessary to have the specified clock association. If `dest` is not null, the result is placed in `dest` and returned. Otherwise, a new object is allocated for the result. This method is the implementation of the abstract method of the `HighResolutionTime` base class. No conversion into `AbsoluteTime` is needed in this case. The clock

association of the result is with the `clock` passed as a parameter. If `clock` is null the association is made with the real-time clock.

Overrides: `absolute315` in class `HighResolutionTime314`

Parameters:

`clock` - The `clock` parameter is used only as the new clock association with the result, since no conversion is needed.

`dest` - If `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The copy of `this` in `dest` if `dest` is not null, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

```
public javax.realtime.AbsoluteTime320 add(
    long millis,
    int nanos)
```

Create a new object representing the result of adding `millis` and `nanos` to the values from `this` and normalizing the result. The result will have the same clock association as `this`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`millis` - The number of milliseconds to be added to `this`.

`nanos` - The number of nanoseconds to be added to `this`.

Returns: A new `AbsoluteTime` object whose time is the normalization of `this` plus `millis` and `nanos`.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 add(
    long millis,
    int nanos,
    javax.realtime.AbsoluteTime320 dest)
```

Return an object containing the value resulting from adding `millis` and `nanos` to the values from `this` and normalizing the result. If `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as

`this`, and the clock association with `dest` is ignored. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`millis` - The number of milliseconds to be added to `this`.

`nanos` - The number of nanoseconds to be added to `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` plus `millis` and `nanos` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 add(
    javax.realtime.RelativeTime332 time)
```

Create a new instance of `AbsoluteTime` representing the result of adding `time` to the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to add to `this`.

Returns: A new `AbsoluteTime` object whose time is the normalization of `this` plus the parameter `time`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 add(
    javax.realtime.RelativeTime332 time,
    javax.realtime.AbsoluteTime320 dest)
```

Return an object containing the value resulting from adding `time` to the value of `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to add to `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` plus the `RelativeTime` parameter `time` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public java.util.Date getDate()
```

Convert the time given by `this` to a `java.util.Date` format. Note that `java.util.Date` represents time as milliseconds so the nanoseconds of `this` will be lost. An `UnsupportedOperationException` is thrown if the `clock` associated with `this` does not have the concept of date.

Returns: A newly allocated `java.util.Date` object with a value of the time past the Epoch represented by `this`.

Throws:

`java.lang.UnsupportedOperationException` - Thrown if the `clock` associated with `this` does not have the concept of date.

```
public javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock)
```

Convert the time of `this` to a relative time, using the given instance of `Clock352` to determine the current time. The calculation is the current time indicated by the given instance of `Clock352` subtracted from the time given by `this`. If `clock` is `null` the real-time clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `clock` passed as a parameter.

Overrides: `relative317` in class `HighResolutionTime314`

Parameters:

`clock` - The instance of `Clock352` used to convert the time of `this` into relative time, and the new clock association for the result.

Returns: The `RelativeTime` conversion in a newly allocated object, associated with the `clock` parameter.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock,
    javax.realtime.RelativeTime332 dest)
```

Convert the time of `this` to a relative time, using the given instance of `Clock352` to determine the current time. The calculation is the current time indicated by the given instance of `Clock352` subtracted from the time given by `this`. If `clock` is `null` the real-time clock is assumed. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock association of the result is with the `clock` passed as a parameter.

Overrides: `relative317` in class `HighResolutionTime314`

Parameters:

`clock` - The instance of `Clock352` used to convert the time of `this` into relative time, and the new clock association for the result.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The `RelativeTime` conversion in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public void set(java.util.Date date)
```

Change the time represented by `this` to that given by the parameter. Note that `java.util.Date` represents time as milliseconds so the nanoseconds of `this` will be set to 0. An `IllegalArgumentException` is thrown if the parameter `date` is `null`. The clock association is implicitly made with the real-time clock.

Parameters:

`date` - A reference to a `java.util.Date` which will become the time represented by `this` after the completion of this method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the parameter `date` is `null`.

```
public javax.realtime.RelativeTime332 subtract(
    javax.realtime.AbsoluteTime320 time)
```

Create a new instance of `RelativeTime` representing the result of subtracting time from the value of `this` and normalizing the result. The clock associated with `this` and the clock associated with the `time` parameter must be the same, and such association is used for the result. An `IllegalArgumentException` is thrown if the clock associated with `this` and the clock associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

Returns: A new `RelativeTime` object whose time is the normalization of `this` minus the `AbsoluteTime` parameter `time`.

Throws:

`java.lang.IllegalArgumentException` - if the clock associated with `this` and the clock associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 subtract(
    javax.realtime.AbsoluteTime320 time,
    javax.realtime.RelativeTime332 dest)
```

Return an object containing the value resulting from subtracting `time` from the value of `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` minus the `AbsoluteTime` parameter `time` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.IllegalArgumentException` - if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 subtract(
    javax.realtime.RelativeTime332 time)
```

Create a new instance of `AbsoluteTime` representing the result of subtracting `time` from the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

Returns: A new `AbsoluteTime` object whose time is the normalization of `this` minus the parameter `time`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 subtract(
    javax.realtime.RelativeTime332 time,
    javax.realtime.AbsoluteTime320 dest)
```

Return an object containing the value resulting from subtracting `time` from the value of `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` minus the `RelativeTime` parameter `time` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public java.lang.String toString()
```

Create a printable string of the time given by `this`.

The string shall be a decimal representation of the milliseconds and nano-second values; formatted as follows “(2251 ms, 750000 ns)”

Overrides: `toString` in class `Object`

Returns: String object converted from the time given by `this`.

9.3 RelativeTime

Declaration

```
public class RelativeTime extends HighResolutionTime314
```

All Implemented Interfaces: `java.lang.Cloneable`, `java.lang.Comparable`

Direct Known Subclasses: `RationalTime341`

Description

An object that represents a time interval $\text{milliseconds}/10^3 + \text{nanoseconds}/10^9$ seconds long. It generally is used to represent a time relative to now.

The time interval is kept in normalized form. The range goes from $[(-2^{63}) \text{ milliseconds} + (-10^6+1) \text{ nanoseconds}]$ to $[(2^{63}-1) \text{ milliseconds} + (10^6-1) \text{ nanoseconds}]$.

A negative interval relative to now represents time in the past. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

9.3.1 Constructors

```
public RelativeTime()
```

Equivalent to `new RelativeTime(0,0)`.

The clock association is implicitly made with the real-time clock.

```
public RelativeTime(javax.realtime.Clock352 clock)
```

Equivalent to `new RelativeTime(0,0,click)`.

The clock association is made with the `clock` parameter. If `clock` is null the association is made with the real-time clock.

Parameters:

`clock` - The clock providing the association for the newly constructed object.

Since: 1.0.1

```
public RelativeTime(long millis, int nanos)
```

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 `IllegalArgumentException` will be thrown.

The clock association is implicitly made with the real-time clock.

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` - Thrown if there is an overflow in the millisecond component when normalizing.

```
public RelativeTime(
    long millis,
    int nanos,
    javax.realtime.Clock352 clock)
```

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 `IllegalArgumentException` will be thrown.

The clock association is made with the `clock` parameter. If `clock` is null the association is made with the real-time clock.

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` - Thrown if there is an overflow in the millisecond component when normalizing.

Since: 1.0.1

```
public RelativeTime(javax.realtime.RelativeTime332 time)
```

Make a new `RelativeTime` object from the given `RelativeTime` object.
The new object will have the same clock association as the `time` parameter.

Parameters:

`time` - The `RelativeTime` object which is the source for the copy.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `time` parameter is null.

```
public RelativeTime(  
    javax.realtime.RelativeTime332 time,  
    javax.realtime.Clock352 clock)
```

Make a new `RelativeTime` object from the given `RelativeTime` object.
The clock association is made with the `clock` parameter. If `clock` is null the association is made with the real-time clock.

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` - Thrown if the `time` parameter is null.

Since: 1.0.1

9.3.2 Methods

```
public javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock)
```

Convert the time of `this` to an absolute time, using the given instance of `Clock352` to determine the current time. The calculation is the current time indicated by the given instance of `Clock352` plus the interval given by `this`. If `clock` is `null` the real-time clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `clock` passed as a parameter.

Overrides: `absolute314` in class `HighResolutionTime314`

Parameters:

`clock` - The instance of `Clock352` used to convert the time of `this` into absolute time, and the new clock association for the result.

Returns: The `AbsoluteTime` conversion in a newly allocated object, associated with the `clock` parameter.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock,
    javax.realtime.AbsoluteTime320 dest)
```

Convert the time of `this` to an absolute time, using the given instance of `Clock352` to determine the current time. The calculation is the current time indicated by the given instance of `Clock352` plus the interval given by `this`. If `clock` is `null` the real-time clock is assumed. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock association of the result is with the `clock` passed as a parameter.

Overrides: `absolute315` in class `HighResolutionTime314`

Parameters:

`clock` - The instance of `Clock352` used to convert the time of `this` into absolute time, and the new clock association for the result.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The `AbsoluteTime` conversion in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 add(
    long millis,
    int nanos)
```

Create a new object representing the result of adding `millis` and `nanos` to the values from `this` and normalizing the result. The result will have the same clock association as `this`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

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` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 add(
    long millis,
    int nanos,
    javax.realtime.RelativeTime332 dest)
```

Return an object containing the value resulting from adding `millis` and `nanos` to the values from `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as `this`, and the clock association with `dest` is ignored. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`millis` - The number of milliseconds to be added to `this`.

`nanos` - The number of nanoseconds to be added to `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` plus `millis` and `nanos` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 add(
    javax.realtime.RelativeTime332 time)
```

Create a new instance of `RelativeTime` representing the result of adding `time` to the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to add to `this`.

Returns: A new `RelativeTime` object whose time is the normalization of `this` plus the parameter `time`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 add(
    javax.realtime.RelativeTime332 time,
    javax.realtime.RelativeTime332 dest)
```

Return an object containing the value resulting from adding `time` to the value of `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the

`time` parameter are expected to be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to add to `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` plus the `RelativeTime` parameter `time` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public void addInterarrivalTo(
    javax.realtime.AbsoluteTime320 timeAndDestination)
```

Deprecated. As of RTSJ 1.0.1

Add the interval of `this` to the given instance of `AbsoluteTime320`.

Parameters:

`timeAndDestination` - A reference to the given instance of `AbsoluteTime320` and the result.

```
public javax.realtime.RelativeTime332 getInterarrivalTime()
```

Deprecated. As of RTSJ 1.0.1

Gets the interval defined by `this`. For an instance of `RationalTime341` it is the interval divided by the frequency.

Returns: A reference to a new instance of `RelativeTime332` with the same interval as `this`.

```
public javax.realtime.RelativeTime332 getInterarrivalTime(
    javax.realtime.RelativeTime332 destination)
```

Deprecated. As of RTSJ 1.0.1

Gets the interval defined by this. For an instance of [RationalTime₃₄₁](#) it is the interval divided by the frequency.

Parameters:

destination - A reference to the new object holding the result.

Returns: A reference to an object holding the result.

```
public javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock)
```

Return a copy of this. A new object is allocated for the result. This method is the implementation of the abstract method of the [HighResolutionTime](#) base class. No conversion into [RelativeTime](#) is needed in this case. The clock association of the result is with the `clock` passed as a parameter. If `clock` is null the association is made with the real-time clock.

Overrides: [relative₃₁₇](#) in class [HighResolutionTime₃₁₄](#)

Parameters:

clock - The `clock` parameter is used only as the new clock association with the result, since no conversion is needed.

Returns: The copy of this in a newly allocated [RelativeTime](#) object, associated with the `clock` parameter.

```
public javax.realtime.RelativeTime332 relative(
    javax.realtime.Clock352 clock,
    javax.realtime.RelativeTime332 dest)
```

Return a copy of this. If `dest` is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. This method is the implementation of the abstract method of the [HighResolutionTime](#) base class. No conversion into [RelativeTime](#) is needed in this case. The clock association of the result is with the `clock` passed as a parameter. If `clock` is null the association is made with the real-time clock.

Overrides: [relative₃₁₇](#) in class [HighResolutionTime₃₁₄](#)

Parameters:

clock - The `clock` parameter is used only as the new clock association with the result, since no conversion is needed.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: The copy of `this` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `clock` parameter.

```
public javax.realtime.RelativeTime332 subtract(
    javax.realtime.RelativeTime332 time)
```

Create a new instance of `RelativeTime` representing the result of subtracting `time` from the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result. An `IllegalArgumentException` is thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

Returns: A new `RelativeTime` object whose time is the normalization of `this` minus the parameter `time` parameter `time`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public javax.realtime.RelativeTime332 subtract(
    javax.realtime.RelativeTime332 time,
    javax.realtime.RelativeTime332 dest)
```

Return an object containing the value resulting from subtracting the value of `time` from the value of `this` and normalizing the result. If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored. An `IllegalArgumentException` is thrown if the

clock associated with `this` and the clock associated with the `time` parameter are different. An `IllegalArgumentException` is thrown if the `time` parameter is `null`. An `ArithmeticException` is thrown if the result does not fit in the normalized format.

Parameters:

`time` - The time to subtract from `this`.

`dest` - If `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns: the result of the normalization of `this` minus the `RelativeTime` parameter `time` in `dest` if `dest` is not `null`, otherwise the result is returned in a newly allocated object.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the if the clock associated with `this` and the clock associated with the `time` parameter are different, or when the `time` parameter is `null`.

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

```
public java.lang.String toString()
```

Create a printable string of the time given by `this`.

The string shall be a decimal representation of the milliseconds and nano-second values; formatted as follows “(2251 ms, 750000 ns)”

Overrides: `toString` in class `Object`

Returns: String object converted from the time given by `this`.

9.4 RationalTime

Declaration

```
public class RationalTime extends RelativeTime332
```

All Implemented Interfaces: `java.lang.Cloneable`, `java.lang.Comparable`

Description

Deprecated. As of RTSJ 1.0.1

An object that represents a time interval $\text{milliseconds}/10^3 + \text{nanoseconds}/10^9$ seconds long that is divided into subintervals by some frequency. This is generally used in periodic events, threads, and feasibility analysis to specify periods where there is a basic period that must be adhered to strictly (the interval), but within that interval the periodic events are supposed to happen frequency times, as uniformly spaced as possible, but clock and scheduling jitter is moderately acceptable.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level. All Implemented Interfaces: `java.lang.Comparable`

9.4.1 Constructors

public RationalTime(int frequency)

Deprecated. 1.0.1

Construct an instance of `RationalTime`. Equivalent to `new RationalTime(1000, 0, frequency)`—essentially a cycles-per-second value.

Throws:

`java.lang.IllegalArgumentException` - Thrown if frequency is less than or equal to zero.

public RationalTime(int frequency, long millis, int nanos)

Deprecated. 1.0.1

Construct an instance of `RationalTime`. All arguments must be greater than or equal to zero.

Parameters:

`frequency` - The frequency value.

`millis` - The milliseconds value.

`nanos` - The nanoseconds value.

Throws:

`java.lang.IllegalArgumentException` - If any of the argument values are less than zero, or if frequency is equal to zero.

```
public RationalTime(
    int frequency,
    javax.realtime.RelativeTime332 interval)
```

Deprecated. 1.0.1

Construct an instance of RationalTime from the given RelativeTime332 .

Parameters:

frequency - The frequency value.

interval - The given instance of RelativeTime332 .

Throws:

java.lang.IllegalArgumentException - If either of the argument values are less than zero, or if frequency is equal to zero.

9.4.2 Methods

```
public javax.realtime.AbsoluteTime320 absolute(
    javax.realtime.Clock352 clock,
    javax.realtime.AbsoluteTime320 destination)
```

Deprecated. 1.0.1

Convert time of this to an absolute time.

Overrides: absolute335 in class RelativeTime332

Parameters:

clock - The reference clock. If null, Clock.getRealTimeClock() is used.

destination - A reference to the destination instance.

```
public void addInterarrivalTo(
    javax.realtime.AbsoluteTime320 destination)
```

Deprecated. 1.0.1

Add the time of this to an AbsoluteTime320 It is almost the same dest.add(this, dest) except that it accounts for (i.e. divides by) the frequency.

Overrides: addInterarrivalTo338 in class RelativeTime332

Parameters:

destination - A reference to the destination instance.

```
public int getFrequency()
```

Deprecated. 1.0.1

Gets the value of frequency.

Returns: The value of frequency as an integer.

```
public javax.realtime.RelativeTime332 getInterarrivalTime()
```

Deprecated. 1.0.1

Gets the interarrival time. This time is $(\text{milliseconds}/10^3 + \text{nanoseconds}/10^9)/\text{frequency}$ rounded down to the nearest expressible value of the fields and their types of [RelativeTime332](#).

Overrides: [getInterarrivalTime338](#) in class [RelativeTime332](#)

```
public javax.realtime.RelativeTime332 getInterarrivalTime(  
    javax.realtime.RelativeTime332 dest)
```

Deprecated. 1.0.1

Gets the interarrival time. This time is $(\text{milliseconds}/10^3 + \text{nanoseconds}/10^9)/\text{frequency}$ rounded down to the nearest expressible value of the fields and their types of [RelativeTime332](#).

Overrides: [getInterarrivalTime339](#) in class [RelativeTime332](#)

Parameters:

dest - Result is stored in dest and returned, if null, a new object is returned.

```
public void set(long millis, int nanos)
```

Deprecated. 1.0.1

Sets the indicated fields to the given values.

Overrides: [set319](#) in class [HighResolutionTime314](#)

Parameters:

millis - The new value for the millisecond field.

nanos - The new value for the nanosecond field.

```
public void setFrequency(int frequency)
```

Deprecated. 1.0.1

Sets the value of the frequency field.

Parameters:

frequency - The new value for the frequency.

Throws:

`java.lang.IllegalArgumentException` - Thrown if frequency is less than or equal to zero.

public java.lang.String toString()

Deprecated. 1.0.1

Create a printable string of the time given by `this`.

The string shall be a decimal representation of the frequency, milliseconds and nanosecond values; formatted as follows “(100, 2251 ms, 750000 ns)”

Overrides: [toString₃₄₁](#) in class [RelativeTime₃₃₂](#)

Returns: String object converted from the time given by `this`.

Unofficial

Unofficial

Chapter 10

Clocks and Timers

This section describes the RTSJ clocks and timers. These classes:

- Allow creation of a timer whose expiration is either periodic or set to occur at a particular time as kept by a system-dependent time base (clock).
- Trigger some behavior to occur on expiration of a timer, using the asynchronous event mechanisms provided by the specification.

Definitions

The following terms and abbreviations will be used:

A *timing mechanism* is either a Clock or a Timer, capable of representing and following the progress of time, by means of time values.

A *monotonic clock* is a clock whose time values are monotonic, and a *monotonic non-decreasing clock* is a clock whose time values are monotonic non-decreasing. Monotonicity is a boolean property, while time synchronization, uniformity and accuracy are characteristics that depend on agreed tolerances.

Time synchronization is a relation between two clocks. Two clocks are synchronized when the difference between their time values is less than some specified offset. Synchronization in general degrades with time, and may be lost, given a specified offset.

Resolution is the minimal time value interval that can be distinguished by a timing mechanism.

Uniformity, in this context, refers to the measurement of the progress of time at a consistent rate, with a tolerance on the variability. Uniformity is affected by two other factors, *jitter* and *stability*.

Jitter is a short-term and non-cumulative small variation caused by noise sources, like thermal noise. More practically, jitter refers to the distribution of the differences between when events are actually fired or noticed by the software and when they should have really occurred according to time in the real-world.

(Lack-of) *stability* accounts for large and often cumulative variations, due to e.g. supply voltage and temperature.

In practice a timing mechanism is driven by an oscillator. *Accuracy* is the difference between the desired frequency and the actual frequency of the oscillator, and a major reason of synchronization loss.

The system *real-time clock* is monotonic non-decreasing, with the epoch as the Epoch (January 1, 1970, 00:00:00 GMT). The system real-time clock is not necessarily synchronized with the external world, and the correctness of the epoch as a time base depends on such synchronization. It is as uniform and accurate as allowed by the underlying hardware.

A `Timer` measures time using a given `Clock`, and fires an event as scheduled.

In the context of a `Timer`, *firing* is the action that causes the release of any associated `AsyncEventHandler`, while the *skipping* of firing is an action with no consequences.

A `Timer` is *active* if it has been started and not stopped since last started and it has a time in the future at which it is expected to fire or skip, else it is *not-active*.

In the context of a `Timer`, *enabling* allows firing, while *disabling* causes the skipping of firing. Enabling and disabling act as a mask over firing.

The behavior of a `OneShotTimer` is that of a `Timer` that does not automatically reschedule its firing after an initial firing, or after the skipping of an initial firing (if *disabled* and *active* when the firing was due). It is specified using an initial firing time.

The behavior of a `PeriodicTimer` is that of a `Timer` that automatically reschedules its firing after an initial firing, or after the skipping of an initial firing if disabled when the firing was due. It is specified using an initial firing time and an interval or period used for the self-rescheduling.

Every timer has at least one clock associated with it, on which the measurement of time will be based. The `OneShotTimer` timer has one, the clock associated with the specification of the initial firing time. The `PeriodicTimer` timer has two, the clock associated with the specification of the initial firing time and the clock associated with the specification of the interval or period.

For both timers it is possible to provide an explicit clock to override the clock associated with parameters. See the individual classes for details.

The *counting time* is the time accumulated while *active* by a `Timer` created or rescheduled using a `RelativeTime` to specify the initial firing/skipping time. The *counting time* is zeroed at the beginning of an activation, or when rescheduled, while *active*, before the initial firing/skipping of an activation.

Overview

The classes provided by this section are `Clock`, `Timer`, `PeriodicTimer`, and `OneShotTimer`.

At least one instance of the abstract `Clock` class is provided by the implementation, the system *real-time clock*, and this instance is made available as a singleton.

`Timer` is an abstract class and consequently only its subclasses can be instantiated. The `Timer` class provides the interface and underlying implementation for both one-shot and periodic timers.

Instances of `OneShotTimer` and `PeriodicTimer` can be created and rescheduled specifying the initial firing time either as an `AbsoluteTime`, to be considered at the application of the start command, or as a `RelativeTime`, to be considered from the application of the start command.

The period for a `PeriodicTimer` is always specified as a `RelativeTime`.

An instance of `OneShotTimer` describes an event that is to be triggered at most once (unless restarted after expiration). It may be used as the source for time-outs.

An instance of `PeriodicTimer` fires or skips on a periodic schedule.

Semantics and Requirements

This list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. Clocks and Timers relation:
 - a. The relation between clocks and timers described below is better understood if a clock is seen as providing an interrupt at intervals based on its resolution.
 - b. Timers can be modeled as counters, or as comparators.

- i. Timer modeled as counter: The timer is viewed as if every clock interrupt increments a count up to the firing count, initially given by either an instance of `RelativeTime` or computed as the difference between an instance of `AbsoluteTime` and a semantically specified “now” (using the same clock).
 - In this model, `start` is understood as defining “now” and start counting, `stop` is understood as stop counting. `start` after `stop` may be understood as start counting again from where stopped, or start from scratch after resetting the count. In both cases a delay is introduced.
 - An `RTSJ Timer`, when using the counter model, resets the count when it is re-started after being stopped.
 - ii. Timer modeled as comparator: The timer is viewed as if every clock interrupt forces a comparison between an absolute time and a firing time, initially given either as an instance of `AbsoluteTime` or computed as the sum of an instance of `RelativeTime` and a semantically specified “now” (using the same clock).
 - In this model, `start` is understood as start comparing, and possibly the first `start` is understood as defining “now”. `stop` is understood as stop comparing. `start` after `stop` may be understood as start comparing again. In this case no delay is introduced.
 - c. When a `Timer` is created or rescheduled using a `RelativeTime` to specify the initial firing time, the `RTSJ` keeps the specified initial firing time as a `RelativeTime` and behaves according to the counter model.
 - d. When a `Timer` is created or rescheduled using an `AbsoluteTime` to specify the initial firing time, the `RTSJ` keeps the specified initial firing time as an `AbsoluteTime` and uses the comparator model.
2. The `Clock` class shall be capable of reporting the achievable resolution of timers based on that clock.
 3. The `OneShotTimer` class shall ensure that a one-shot timer is triggered at most once unless restarted after expiration.
 4. The `PeriodicTimer` class shall allow the period of the timer to be expressed in terms of a `RelativeTime` or a `RationalTime` (now deprecated).
 5. The initial firing, or skipping, of a `PeriodicTimer` occurs in response to the invocation of its `start` method, in accordance with the start time passed to its constructor. This initial firing, or skipping, may be rescheduled by a call to the `reschedule` method, in accordance with the time passed to that method.
 6. For a `PeriodicTimer`:

- a. Let S be the absolute time at which the initial firing or skipping of a `PeriodicTimer` is scheduled to occur:
 - i. If the start, or reschedule, time was given as an absolute time, A , and that time is in the future when the timer is made active, then S equals A . Otherwise, if the absolute time has passed when the timer is made active, then S equals the time at which the timer was made active.
 - ii. If the start, or reschedule, time was given as a relative time, R , then S equals the time at which the *counting time*, started when the timer was made *active*, equals R .
NOTE: The transition to *not-active* by this timer causes the *counting time* to reset, effectively preventing this kind of timer from firing immediately, unless given a time value of 0.
 - b. The firings, or skipings, of a `PeriodicTimer` are scheduled to occur according to $S+nT$, for $n = 0, 1, 2 \dots$, where S is as just specified, and T is the interval of the periodic timer.
7. When in a *not-active* state a `Timer` retains the parameters given at construction time or the parameters it had at de-activation time. Those are the parameters that will be used upon invocation of `start` while in that state, unless the parameters are explicitly changed before that, using `reschedule` and `setInterval` as appropriate.
 8. If a `Timer` object is allocated in a scoped memory area, then it will increment the reference count associated with that area. Such a reference count will only be decremented when the `Timer` object is destroyed. (See semantics in the *Memory* chapter for details.)
 9. A `Timer` object will not fire before its due time.

Rationale

Clocks differ because of monotonicity, synchronization, jitter, stability, accuracy, and resolution. There are many possible subclasses of clocks: real-time clocks, user time clocks, simulation time clocks, wall clocks.

The idea of using multiple clocks may at first seem unusual but it is allowed to accommodate differences and as a possible resource allocation strategy. Consider a real-time system where the natural events of the system have different tolerances for jitter. Assume the system functions properly if event A is fired within plus or minus 100 seconds of the actual time it should occur but event B must be fired within 100 microseconds of its actual time. Further assume, without loss of generality, that events A and B are periodic. An application could then create two instances of

`PeriodicTimer` based on two clocks. The timer for event *B* should be based on a `Clock` which checks its queue at least every 100 microseconds but the timer for event *A* could be based on a `Clock` that checked its queue only every 100 seconds. This use of two clocks reduces the queue size of the accurate clock and thus queue management overhead is reduced.

The importance of the use of one-shot timers for time-out behavior and the vagaries in the execution of code prior to starting the timer for short time-outs dictate that the triggering of the timer should be guaranteed. The problem is exacerbated for periodic timers where the importance of the periodic triggering outweighs the precision of the start time. In such cases, it is also convenient to allow, for example, a relative time of zero to be used as the start time.

10.1 Clock

Declaration

```
public abstract class Clock
```

Description

A clock marks the passing of time. It has a concept of now that can be queried through `Clock.getTime()`, and it can have events queued on it which will be fired when their appointed time is reached.

10.1.1 Constructors

```
public Clock()
```

Constructor for the abstract class.

10.1.2 Methods

```
public abstract javax.realtime.RelativeTime332
    getEpochOffset()
```

Returns the relative time of the offset of the epoch of `this` clock from the Epoch. For the real-time clock it will return a `RelativeTime` value equal to 0. An `UnsupportedOperationException` is thrown if the clock does not support the concept of date.

Returns: A newly allocated `RelativeTime332` object in the current execution context with the offset past the Epoch for `this` clock. The returned object is associated with `this` clock.

Throws:

`java.lang.UnsupportedOperationException` - Thrown if the clock does not have the concept of date.

Since: 1.0.1

```
public static javax.realtime.Clock352 getRealtimeClock()
```

There is always at least one clock object available: the system real-time clock. This is the default `Clock`.

Returns: The singleton instance of the default `Clock`

```
public abstract javax.realtime.RelativeTime332
    getResolution()
```

Gets the resolution of the clock, the nominal interval between ticks.

Returns: A newly allocated `RelativeTime332` object in the current execution context representing the resolution of `this`. The returned object is associated with `this` clock.

```
public abstract javax.realtime.AbsoluteTime320 getTime()
```

Gets the current time in a newly allocated object.

Note: This method will return an absolute time value that represents the clock's notion of an absolute time. For clocks that do not measure calendar time this absolute time may not represent a wall clock time.

Returns: A newly allocated instance of `AbsoluteTime320` in the current allocation context, representing the current time. The returned object is associated with `this` clock.

```
public abstract javax.realtime.AbsoluteTime320 getTime(
    javax.realtime.AbsoluteTime320 dest)
```

Gets the current time in an existing object. The time represented by the given `AbsoluteTime320` is changed at some time between the invocation of the method and the return of the method. *Note:* This method will return an absolute time value that represents the clock's notion of an absolute time. For clocks that do not measure calendar time this absolute time may not represent a wall clock time.

Parameters:

`dest` - The instance of `AbsoluteTime320` object which will be updated in place. The clock association of the `dest` parameter is ignored. When `dest` is not `null` the returned object is associated with this clock. If `dest` is `null`, then nothing happens.

Returns: The instance of `AbsoluteTime320` passed as parameter, representing the current time, associated with this clock, or `null` if `dest` was `null`.

Since: 1.0.1 The return value is updated from `void` to `AbsoluteTime`.

```
public abstract void setResolution(
    javax.realtime.RelativeTime332 resolution)
```

Set the resolution of `this`. For some hardware clocks setting resolution is impossible and if this method is called on those clocks, then an `UnsupportedOperationException` is thrown.

Parameters:

`resolution` - The new resolution of `this`, if the requested value is supported by this clock. If `resolution` is smaller than the minimum resolution supported by this clock then it throws `IllegalArgumentException`. If the requested resolution is not available and it is larger than the minimum resolution, then the clock will be set to the closest resolution that the clock supports, via truncation. The value of the `resolution` parameter is not altered. The clock association of the `resolution` parameter is ignored.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `resolution` is `null`, or if the requested resolution is smaller than the minimum resolution supported by this clock.

`java.lang.UnsupportedOperationException` - Thrown if the clock does not support setting its resolution.

10.2 Timer

Declaration

```
public abstract class Timer extends AsyncEvent388
```

Direct Known Subclasses: `OneShotTimer369`, `PeriodicTimer371`

Description

A *timer* is a timed event that measures time according to a given `Clock352`. This class defines basic functionality available to all timers. Applications will generally use either `PeriodicTimer371` to create an event that is fired repeatedly at regular intervals, or `OneShotTimer369` for an event that just fires once at a specific time. A timer is always associated with at least one `Clock352`, which provides the basic facilities of something that ticks along following some time line (real-time, CPU-time, user-time, simulation-time, etc.). All timers are created *disabled* and do nothing until `start()` is called.

Pseudo-Code Representation of State Transitions for Timer

An implementation shall behave effectively as if it implemented the following pseudo-code. Only absolute and relative time behaviors are shown as rational time has been deprecated.

NOTE: The pseudo-code does not take into account any issue of synchronization, it just shows the functionality, and the intended behavior is obtained with groups of and'ed statements interpreted as atomic. This is relevant, for example, in cases where the *firing* of an `AsyncEventHandler393` is part of the statements preceding a state transition. While the *firing* causes the release of the handler before the state transition, the execution of the handler does not take place until after the state transition has completed.

The pseudo-code is a model, it should be interpreted as running continuously, with instructions that take no time.

```
absolute construction state is {not-active, disabled, absolute}
with nextTargetTime = absoluteTime
last_rescheduled_with_AbsoluteTime = TRUE
[(if PeriodicTimer) period = interval]
```

```
relative construction state is {not-active, disabled, relative}
with nextDurationTime = relativeTime
last_rescheduled_with_AbsoluteTime = FALSE
[(if PeriodicTimer) period = interval]
```

```
{not-active, disabled, absolute}
[(if PeriodicTimer)
  set fired_or_skipped_in_current_activation = FALSE]
enable -> no state change, do nothing
disable -> no state change, do nothing
stop -> no state change, return FALSE
start ->
  [if last_rescheduled_with_AbsoluteTime
    then
      [set targetTime = nextTargetTime
        [if targetTime < currentTime
          then set targetTime = currentTime]
        then go to state {active, enabled, absolute}]
    else
      [set countingTime = 0
        and set durationTime = nextDurationTime]
```

```

        then go to state {active, enabled, relative}]
isRunning -> return FALSE
reschedule ->
    [if using an instance of AbsoluteTime
    then
        [reset the nextTargetTime to absoluteTime arg
        and set last_rescheduled_with_AbsoluteTime = TRUE
        and no state change]
    else
        [reset the nextDurationTime to relativeTime arg
        and set last_rescheduled_with_AbsoluteTime = FALSE
        and go to state {not-active, disabled, relative}]
getFireTime -> throws IllegalStateException
destroy -> go to state {destroyed}
startDisabled ->
    [if last_rescheduled_with_AbsoluteTime
    then
        [set targetTime = nextTargetTime
        [if targetTime < currentTime
        then set targetTime = currentTime]
        then go to state {active, disabled, absolute}]
    else
        [set countingTime = 0
        and set durationTime = nextDurationTime
        then go to state {active, disabled, relative}]]

{not-active, disabled, relative}
[[if PeriodicTimer)
    set fired_or_skipped_in_current_activation = FALSE]
enable -> no state change, do nothing
disable -> no state change, do nothing
stop -> no state change, return FALSE
start ->
    [if last_rescheduled_with_AbsoluteTime
    then
        [set targetTime = nextTargetTime
        [if targetTime < currentTime
        then set targetTime = currentTime]
        then go to state {active, enabled, absolute}]
    else
        [set countingTime = 0
        and set durationTime = nextDurationTime
        then go to state {active, enabled, relative}]]
isRunning -> return FALSE
reschedule ->
    [if using an instance of AbsoluteTime
    then
        [reset the nextTargetTime to absoluteTime arg
        and set last_rescheduled_with_AbsoluteTime = TRUE
        and go to state {not-active, disabled, absolute}]
    else
        [reset the nextDurationTime to relativeTime arg
        and set last_rescheduled_with_AbsoluteTime = FALSE
        and no state change]]
getFireTime -> throws IllegalStateException
destroy -> go to state {destroyed}
startDisabled ->
    [if last_rescheduled_with_AbsoluteTime
    then
        [set targetTime = nextTargetTime
        [if targetTime < currentTime
        then set targetTime = currentTime]
        then go to state {active, disabled, absolute}]
    else
        [set countingTime = 0
        and set durationTime = nextDurationTime

```

```

then go to state {active, disabled, relative}]

{active, enabled, absolute}
[if currentTime >= targetTime
then
  [if PeriodicTimer
  then
    [if period > 0
    then
      [fire
      and set fired_or_skipped_in_current_activation = TRUE
      and self reschedule
      via targetTime = (targetTime + period)
      and re-enter current state]
    else
      [fire
      and go to state {not-active, disabled, absolute}]]
  else
    [it is a OneShotTimer so
    fire
    and go to state {not-active, disabled, absolute}]]]
enable -> no state change, do nothing
disable -> go to state {active, disabled, absolute}
stop -> [go to state {not-active, disabled, absolute}
and return TRUE]
start -> throws IllegalStateException
isRunning -> return TRUE
reschedule ->
  [if NOT fired_or_skipped_in_current_activation
  then
    [if using an instance of AbsoluteTime
    then
      [reset the targetTime to absoluteTime arg
      and re-enter current state]
    else
      [reset the durationTime to relativeTime arg
      and set countingTime = 0
      and go to state {active, enabled, relative}]]]
  else
    [if using an instance of AbsoluteTime
    then
      [reset the nextTargetTime to absoluteTime arg
      and set last_rescheduled_with_AbsoluteTime = TRUE
      and no state change]
    else
      [reset the nextDurationTime to relativeTime arg
      and set last_rescheduled_with_AbsoluteTime = FALSE
      and no state change]]]
getFireTime -> return targetTime
destroy -> go to state {destroyed}
startDisabled -> throws IllegalStateException

```

```

{active, enabled, relative}
[if countingTime >= durationTime
then
  [if PeriodicTimer
  then
    [if period > 0
    then
      [fire
      and set fired_or_skipped_in_current_activation = TRUE
      and self reschedule
      via durationTime = (durationTime + period)
      and re-enter current state]
    else
      [fire
      and go to state {not-active, disabled, absolute}]]]
  else
    [it is a OneShotTimer so
    fire
    and go to state {not-active, disabled, absolute}]]]
enable -> no state change, do nothing
disable -> go to state {active, disabled, absolute}
stop -> [go to state {not-active, disabled, absolute}
and return TRUE]
start -> throws IllegalStateException
isRunning -> return TRUE
reschedule ->
  [if NOT fired_or_skipped_in_current_activation
  then
    [if using an instance of AbsoluteTime
    then
      [reset the targetTime to absoluteTime arg
      and re-enter current state]
    else
      [reset the durationTime to relativeTime arg
      and set countingTime = 0
      and go to state {active, enabled, relative}]]]
  else
    [if using an instance of AbsoluteTime
    then
      [reset the nextTargetTime to absoluteTime arg
      and set last_rescheduled_with_AbsoluteTime = TRUE
      and no state change]
    else
      [reset the nextDurationTime to relativeTime arg
      and set last_rescheduled_with_AbsoluteTime = FALSE
      and no state change]]]
getFireTime -> return targetTime
destroy -> go to state {destroyed}
startDisabled -> throws IllegalStateException

```

```

        [fire
         and go to state {not-active, disabled, relative}]]
    else
        [it is a OneShotTimer so
         fire
         and go to state {not-active, disabled, relative}]]]
    enable -> no state change, do nothing
    disable -> go to state {active, disabled, relative}
    stop -> [go to state {not-active, disabled, relative}
            and return TRUE]
    start -> throws IllegalStateException
    isRunning -> return TRUE
    reschedule ->
        [if NOT fired_or_skipped_in_current_activation
         then
         [if using an instance of AbsoluteTime
          then
           [reset the targetTime to absoluteTime arg
            and go to state {active, enabled, absolute}]]
          else
           [reset the durationTime to relativeTime arg
            and set countingTime = 0
            and re-enter current state]]]
        else
        [if using an instance of AbsoluteTime
         then
          [reset the nextTargetTime to absoluteTime arg
           and set last_rescheduled_with_AbsoluteTime = TRUE
           and no state change]
         else
          [reset the nextDurationTime to relativeTime arg
           and set last_rescheduled_with_AbsoluteTime = FALSE
           and no state change]]]
    getFireTime ->
        return (currentTime + durationTime - countingTime)
    destroy -> go to state {destroyed}
    startDisabled -> throws IllegalStateException

{active, disabled, absolute}
    [if currentTime >= targetTime
     then
     [if PeriodicTimer
      then
      [if period > 0
       then
       [skip
        and set fired_or_skipped_in_current_activation = TRUE
        and self reschedule
         via targetTime = (targetTime + period)
        and re-enter current state]
       else
       [skip
        and go to state {not-active, disabled, absolute}]]]
      else
      [it is a OneShotTimer so
       skip
       and go to state {not-active, disabled, absolute}]]]
    enable -> go to state {active, enabled, absolute}
    disable -> no state change, do nothing
    stop -> [go to state {not-active, disabled, absolute}
            and return TRUE]
    start -> throws IllegalStateException
    isRunning -> return FALSE
    reschedule ->
        [if NOT fired_or_skipped_in_current_activation
         then

```

```

[if using an instance of AbsoluteTime
 then
  [reset the targetTime to absoluteTime arg
   and re-enter current state]
 else
  [reset the durationTime to relativeTime arg
   and set countingTime = 0
   and go to state {active, disabled, relative}]]
else
 [if using an instance of AbsoluteTime
  then
   [reset the nextTargetTime to absoluteTime arg
    and set last_rescheduled_with_AbsoluteTime = TRUE
    and no state change]
  else
   [reset the nextDurationTime to relativeTime arg
    and set last_rescheduled_with_AbsoluteTime = FALSE
    and no state change]]]
getFireTime -> return targetTime
destroy -> go to state {destroyed}
startDisabled -> throws IllegalStateException

{active, disabled, relative}
[if countingTime >= durationTime
 then
  [if PeriodicTimer
   then
    [if period > 0
     then
      [skip
       and set fired_or_skipped_in_current_activation = TRUE
       and self reschedule
        via durationTime = (durationTime + period)
       and re-enter current state]
     else
      [skip
       and go to state {not-active, disabled, relative}]]]
  else
   [it is a OneShotTimer so
    skip
    and go to state {not-active, disabled, relative}]]]
enable -> go to state {active, enabled, relative}
disable -> no state change, do nothing
stop -> [go to state {not-active, disabled, relative}
        and return TRUE]
start -> throws IllegalStateException
isRunning -> return FALSE
reschedule ->
 [if NOT fired_or_skipped_in_current_activation
  then
   [if using an instance of AbsoluteTime
    then
     [reset the targetTime to absoluteTime arg
      and go to state {active, disabled, absolute}]
    else
     [reset the durationTime to relativeTime arg
      and set countingTime = 0
      and re-enter current state]]]
  else
   [if using an instance of AbsoluteTime
    then
     [reset the nextTargetTime to absoluteTime arg
      and set last_rescheduled_with_AbsoluteTime = TRUE
      and no state change]
    else
     [reset the nextDurationTime to relativeTime arg

```

```

        and set last_rescheduled_with_AbsoluteTime = FALSE
        and no state change]]]
getFireTime ->
  return (currentTime + durationTime - countingTime)
destroy -> go to state {destroyed}
startDisabled -> throws IllegalStateException

```

```

{destroyed}
  enable | disable | stop | start | isRunning
  | reschedule | getFireTime | destroy
  | startDisabled -> throws IllegalStateException

```

The following two methods, without loss of generality and to avoid clutter, have been omitted from the above Pseudo-code.

```

Every state but {destroyed} has:
[(if PeriodicTimer) setInterval -> reset period = interval]
[(if PeriodicTimer) getInterval -> return period]

```

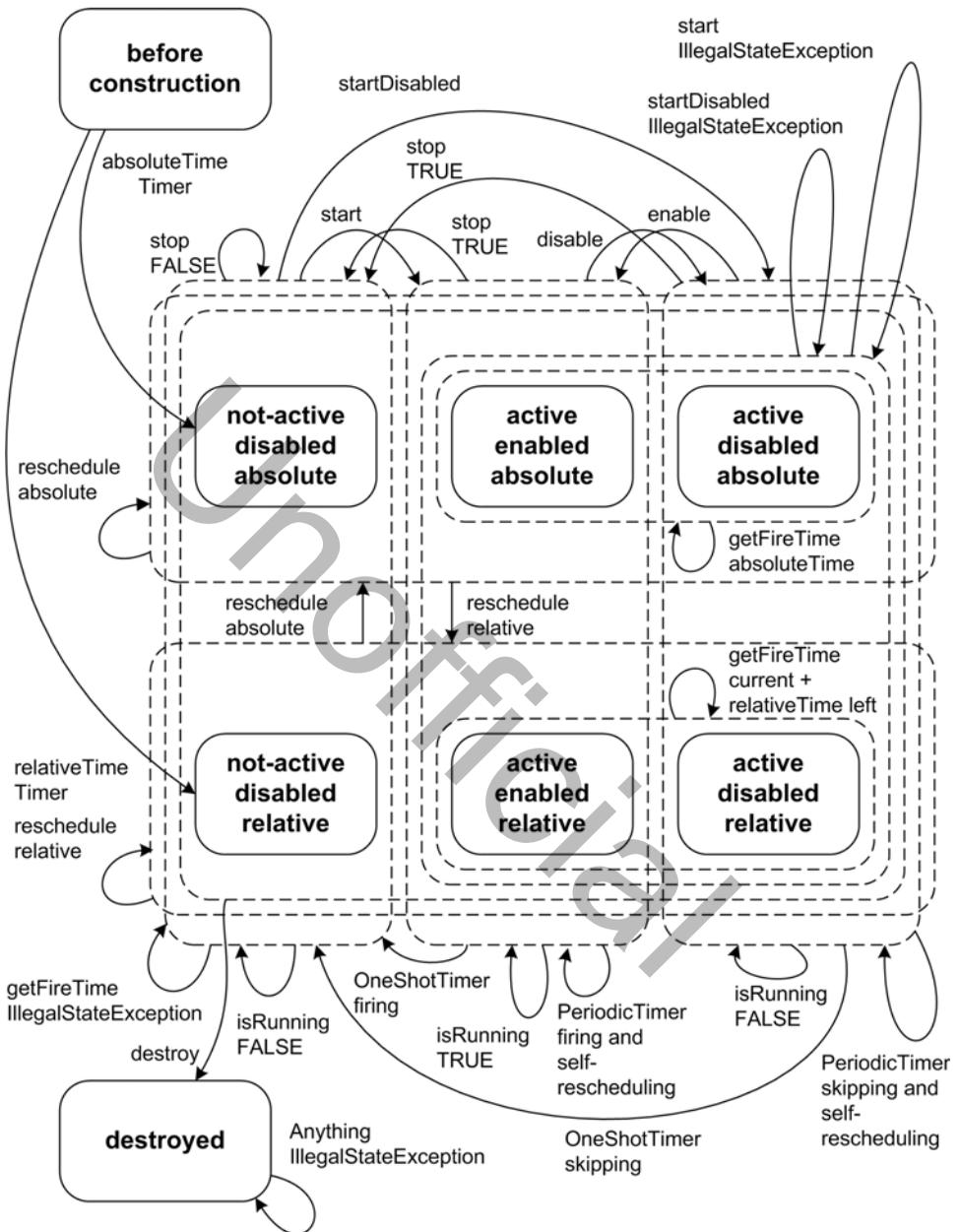
```

The state {destroyed} has:
[(if PeriodicTimer) setInterval -> throws IllegalStateException]
[(if PeriodicTimer) getInterval -> throws IllegalStateException]

```

Compact Graphic Representation of State Transitions for Timer

The following compact graphic representation, while not as detailed, complements the State Transitions for Timer pseudo-code:



10.2.1 Constructors

```
protected Timer(
    javax.realtime.HighResolutionTime314 time,
    javax.realtime.Clock352 clock,
    javax.realtime.AsyncEventHandler393 handler)
```

Create a timer that fires according to the given `time`, based on the `Clock352` `clock` and is handled by the specified `AsyncEventHandler393` `handler`.

Parameters:

`time` - The time used to determine when to fire the event. A `time` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

`clock` - The clock on which to base this timer, overriding the clock associated with the parameter `time`. If `null`, the system `Realtime` clock is used. The clock associated with the parameter `time` is always ignored.

`handler` - The default handler to use for this event. If `null`, no handler is associated with the timer and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `time` is a negative `RelativeTime` value.

`java.lang.UnsupportedOperationException` - Thrown if the timer functionality cannot be supported using the given `clock`.

`IllegalAssignmentError448` - Thrown if this `Timer` cannot hold references to `handler` and `clock`.

10.2.2 Methods

```
public void addHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Description copied from class: `javax.realtime.AsyncEvent388`

Add a handler to the set of handlers associated with this event. An instance of `AsyncEvent` may have more than one associated handler. However, add-

ing a handler to an event has no effect if the handler is already attached to the event.

The execution of this method is atomic with respect to the execution of the `fire()` method.

Since this affects the constraints expressed in the release parameters of an existing schedulable object, this may change the feasibility of the current system. This method does not change feasibility set of any scheduler, and no feasibility test is performed.

Note, there is an implicit reference to the handler stored in `this`. The assignment must be valid under any applicable memory assignment rules.

Overrides: [addHandler₃₈₉](#) in class [AsyncEvent₃₈₈](#)

Throws:

`java.lang.IllegalStateException` - Thrown if this Timer has been *destroyed*.

`IllegalAssignmentError448` - Thrown if this [AsyncEvent](#) cannot hold a reference to handler.

Since: 1.0.1

public void bindTo(`java.lang.String` happening)

Should not be called.

Overrides: [bindTo₃₈₉](#) in class [AsyncEvent₃₈₈](#)

Throws:

`java.lang.UnsupportedOperationException` - Thrown if `bindTo` is called on a Timer.

Since: 1.0.1

**public [javax.realtime.ReleaseParameters₁₁₆](#)
createReleaseParameters()**

Create a [ReleaseParameters₁₁₆](#) object appropriate to the timing characteristics of this event. The default is the most pessimistic:

[AperiodicParameters₁₂₉](#). This is typically called by code that is setting up a handler for this event that will fill in the parts of the release parameters for which it has values, e.g. cost.

Overrides: [createReleaseParameters₃₉₀](#) in class [AsyncEvent₃₈₈](#)

Returns: A newly created [ReleaseParameters₁₁₆](#) object.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

`public void destroy()`

Stop this from counting or comparing if *active*, remove from it all the associated handlers if any, and release as many of its resources as possible back to the system. Every method invoked on a `Timer` that has been *destroyed* will throw `IllegalStateException`.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

`public void disable()`

Disable this timer, preventing it from firing. It may subsequently be *re-enabled*. If the timer is *disabled* when its fire time occurs then it will not fire. However, a *disabled* timer created using an instance of `RelativeTime` for its time parameter continues to count while it is *disabled*, and no changes take place in a *disabled* timer created using an instance of `AbsoluteTime`, in both cases the potential firing is simply masked, or skipped. If the timer is subsequently *re-enabled* before its fire time and it is *enabled* when its fire time occurs, then it will fire. It is important to note that this method does not delay the time before a possible firing. For example, if the timer is set to fire at time 42 and the `disable()` is called at time 30 and `enable()` is called at time 40 the firing will occur at time 42 (not time 52). These semantics imply also that firings are not queued. Using the above example, if `enable` was called at time 43 no firing will occur, since at time 42 this was *disabled*. If the `Timer` is already *disabled*, whether it is *active* or *not-active*, this method does nothing.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

`public void enable()`

Re-enable this timer after it has been *disabled*. (See `disable()`₃₆₄.) If the `Timer` is already *enabled*, this method does nothing. If the `Timer` is *not-active*, this method does nothing.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

public void fire()

Should not be called. The `fire` method is reserved for the use of the timer.

Overrides: `fire390` in class `AsyncEvent388`

Throws:

`java.lang.UnsupportedOperationException` - Thrown if `fire` is called from outside the `Timer` implementation.

Since: 1.0.1 Throws `UnsupportedOperationException` instead of doing unspecified damage.

public `java.xml.realtime.Clock352` getClock()

Return the instance of `Clock352` on which this timer is based.

Returns: The instance of `Clock352` associated with this `Timer`.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

public `java.xml.realtime.AbsoluteTime320` getFireTime()

Get the time at which this `Timer` is expected to fire. If the `Timer` is *disabled*, the returned time is that of the skipping of the firing. If the `Timer` is *not-active* it throws `IllegalStateException`.

Returns: The absolute time at which this is expected to fire or to skip, in a newly allocated `AbsoluteTime320` object. If the timer has been created or re-scheduled (see `reschedule(HighResolutionTime)367`) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. The clock association of the returned time is the clock on which this timer is based.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if it is *not-active*.

```
public javax.realtime.AbsoluteTime320 getFireTime(
    javax.realtime.AbsoluteTime320 dest)
```

Get the time at which this `Timer` is expected to fire. If the `Timer` is *disabled*, the returned time is that of the skipping of the firing. If the `Timer` is *not-active* it throws `IllegalStateException`.

Parameters:

`dest` - The instance of `AbsoluteTime320` which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is `null` a new object is allocated for the result.

Returns: The instance of `AbsoluteTime320` passed as parameter, with time values representing the absolute time at which `this` is expected to fire or to skip. If the `dest` parameter is `null` the result is returned in a newly allocated object. If the timer has been created or re-scheduled (see `reschedule(HighResolutionTime)367`) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. The clock association of the returned time is the clock on which `this` timer is based.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if it is *not-active*.

Since: 1.0.1

```
public boolean handledBy(
    javax.realtime.AsyncEventHandler393 handler)
```

Description copied from class: `javax.realtime.AsyncEvent388`

Test to see if the handler given as the parameter is associated with `this`.

Overrides: `handledBy391` in class `AsyncEvent388`

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

Since: 1.0.1

public boolean **isRunning()**

Tests this to determine if this is *active* and is *enabled* such that when the given time occurs it will fire the event. Given the `Timer` current state it answer the question “*Is firing expected?*”.

Returns: `true` if the timer is *active* and *enabled*; `false`, if the timer has either not been *started*, it has been *started* but it is *disabled*, or it has been *started* and is now *stopped*.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

public void **removeHandler**([javax.realtime.AsyncEventHandler₃₉₃](#) handler)

Description copied from class: [javax.realtime.AsyncEvent₃₈₈](#)

Remove a handler from the set associated with this event. The execution of this method is atomic with respect to the execution of the `fire()` method.

A removed handler continues to execute until its `fireCount` becomes zero and it completes.

Since this affects the constraints expressed in the release parameters of an existing schedulable object, this may change the feasibility of the current system. This method does not change the feasibility set of any scheduler, and no feasibility test is performed.

Overrides: [removeHandler₃₉₁](#) in class [AsyncEvent₃₈₈](#)

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

Since: 1.0.1

public void **reschedule**([javax.realtime.HighResolutionTime₃₁₄](#) time)

Change the scheduled time for this event. This method can take either an `AbsoluteTime` or a `RelativeTime` for its argument, and the `Timer` will behave as if created using that type for its `time` parameter. The rescheduling will take place between the invocation and the return of the method.

NOTE: While the scheduled time is changed as described above, the rescheduling itself is applied only on the first firing (or on the first skipping if *disabled*) of a timer’s activation. If `reschedule` is invoked after the cur-

rent activation timer's firing, then the rescheduled time will be effective only upon the next `start` or `startDisabled` command (which may need to be preceded by a `stop` command).

If `reschedule` is invoked with a `RelativeTime` `time` on an *active* timer before its first firing/skipping, then the rescheduled firing/skipping time is relative to the time of invocation.

Parameters:

`time` - The time to reschedule for this event firing. If `time` is `null`, the previous time is still the time used for the `Timer` firing. The clock associated with the parameter `time` is always ignored.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `time` is a negative `RelativeTime` value.

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

```
public void setHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Description copied from class: [javax.realtime.AsyncEvent₃₈₈](#)

Associate a new handler with this event and remove all existing handlers. The execution of this method is atomic with respect to the execution of the `fire()` method.

Since this affects the constraints expressed in the release parameters of the existing schedulable objects, this may change the feasibility of the current system. This method does not change the feasibility set of any scheduler, and no feasibility test is performed.

Overrides: [setHandler₃₉₂](#) in class [AsyncEvent₃₈₈](#)

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

[IllegalAssignmentError₄₄₈](#) - Thrown if this `AsyncEvent` cannot hold a reference to `handler`.

Since: 1.0.1

```
public void start()
```

Start this timer. A timer starts measuring time from when it is started; this method makes the timer *active* and *enabled*.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if this timer is already *active*.

`public void start(boolean disabled)`

Start this timer. A timer starts measuring time from when it is started. If `disabled` is `true` start the timer making it *active* in a *disabled* state. If `disabled` is `false` this method behaves like the `start()` method.

Parameters:

`disabled` - If `true`, the timer will be *active* but *disabled* after it is started. If `false` this method behaves like the `start()` method.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if this timer is already *active*.

Since: 1.0.1

`public boolean stop()`

Stops a timer if it is *active* and changes its state to *not-active* and *disabled*.

Returns: `true` if this was *active* and `false` otherwise.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

10.3 OneShotTimer

Declaration

```
public class OneShotTimer extends Timer354
```

Description

A timed [AsyncEvent₃₈₈](#) that is driven by a [Clock₃₅₂](#). It will fire off once, when the clock time reaches the time-out time, unless restarted after expiration. If the timer is *disabled* at the expiration of the indicated time, the firing is lost (*skipped*). After expiration, the `OneShotTimer` becomes *not-active* and *disabled*. If the clock time has already passed the time-out time, it will fire immediately after it is started or after it is rescheduled while *active*.

Semantics details are described in the [Timer₃₅₄](#) pseudocode and compact graphic representation of state transitions.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

10.3.1 Constructors

```
public OneShotTimer(
    javax.realtime.HighResolutionTime314 time,
    javax.realtime.AsyncEventHandler393 handler)
```

Create an instance of [OneShotTimer₃₆₉](#), based on the [Clock₃₅₂](#) associated with the `time` parameter, that will execute its `fire` method according to the given time.

Parameters:

`time` - The time used to determine when to fire the event. A `time` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

`handler` - The [AsyncEventHandler₃₉₃](#) that will be released when `fire` is invoked. If `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `time` is a `RelativeTime` instance less than zero.

`java.lang.UnsupportedOperationException` - Thrown if the timer functionality cannot be supported using the `clock` associated with `time`.

[IllegalAssignmentError₄₄₈](#) - Thrown if this `OneShotTimer` cannot hold a reference to `handler`.

```
public OneShotTimer(
    javax.realtime.HighResolutionTime314 time,
    javax.realtime.Clock352 clock,
    javax.realtime.AsyncEventHandler393 handler)
```

Create an instance of `OneShotTimer369`, based on the given `clock`, that will execute its `fire` method according to the given `time`. The `Clock352` association of the parameter `time` is ignored.

Parameters:

`time` - The time used to determine when to fire the event. A `time` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

`clock` - The clock on which to base this timer, overriding the clock associated with the parameter `time`. If `null`, the system `Realtime` `clock` is used. The clock associated with the parameter `time` is always ignored.

`handler` - The `AsyncEventHandler393` that will be released when `fire` is invoked. If `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `time` is a `RelativeTime` instance less than zero.

`java.lang.UnsupportedOperationException` - Thrown if the timer functionality cannot be supported using the given `clock`.

`IllegalAssignmentError448` - Thrown if this `OneShotTimer` cannot hold references to `handler` and `clock`.

10.4 PeriodicTimer

Declaration

```
public class PeriodicTimer extends Timer354
```

Description

An `AsyncEvent388` whose `fire` method is executed periodically according to the given parameters. The beginning of the first period is set or measured using the clock associated with the `Timer` start time. The calculation of the period uses the clock

associated with the `Timer` `interval`, unless a `Clock`₃₅₂ is given, in which case the calculation of the period uses that clock.

The first firing is at the beginning of the first interval.

If an interval greater than 0 is given, the timer will fire periodically. If an interval of 0 is given, the `PeriodicTimer` will only fire once, unless restarted after expiration, behaving like a `OneShotTimer`. In all cases, if the timer is *disabled* when the firing time is reached, that particular firing is lost (*skipped*). If *enabled* at a later time, it will fire at its next scheduled time.

If the clock time has already passed the beginning of the first period, the `PeriodicTimer` will fire immediately after it is started.

If one of the `HighResolutionTime`₃₁₄ argument types is `RationalTime`₃₄₁ (now deprecated) then the system guarantees that the fire method will be executed exactly frequency times every unit time (see `RationalTime` constructors) by adjusting the interval between executions of `fire()`. This is similar to a thread with `PeriodicParameters` except that it is lighter weight.

Semantics details are described in the `Timer`₃₅₄ pseudo-code and compact graphic representation of state transitions.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

10.4.1 Constructors

```
public PeriodicTimer(
    javax.realtime.HighResolutionTime314 start,
    javax.realtime.RelativeTime332 interval,
    javax.realtime.AsyncEventHandler393 handler)
```

Create an instance of `PeriodicTimer`₃₇₁ that executes its fire method periodically.

Parameters:

`start` - The time that specifies when the first interval begins, based on the clock associated with it. A `start` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

`interval` - The period of the timer. Its usage is based on the clock associated with it. If `interval` is zero or `null`, the period is ignored and the firing behavior of the `PeriodicTimer` is that of a `OneShotTimer`₃₆₉.

`handler` - The [AsyncEventHandler₃₉₃](#) that will be released when the timer fires. If `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `start` or `interval` is a `RelativeTime` instance with a value less than zero.

[IllegalAssignmentError₄₄₈](#) - Thrown if this `PeriodicTimer` cannot hold references to `handler` and `interval`.

`java.lang.UnsupportedOperationException` - Thrown if the timer functionality cannot be supported using the `clock` associated with `start` or the `clock` associated with `interval`.

```
public PeriodicTimer(
    javax.realtime.HighResolutionTime314 start,
    javax.realtime.RelativeTime332 interval,
    javax.realtime.Clock352 clock,
    javax.realtime.AsyncEventHandler393 handler)
```

Create an instance of [PeriodicTimer₃₇₁](#) that executes its `fire` method periodically.

Parameters:

`start` - The time that specifies when the first interval begins, based on the clock associated with it. A `start` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

`interval` - The period of the timer. Its usage is based on the clock specified by the `clock` parameter. If `interval` is zero or `null`, the period is ignored and the firing behavior of the `PeriodicTimer` is that of a [OneShotTimer₃₆₉](#).

`clock` - The clock to be used to time the `interval`. If `null`, the system `Realtime` clock is used. The [Clock₃₅₂](#) association of the parameter `interval` is always ignored.

`handler` - The [AsyncEventHandler₃₉₃](#) that will be released when `fire` is invoked. If `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a

handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `start` or `interval` is a `RelativeTime` instance with a value less than zero.

`IllegalAssignmentError448` - Thrown if this `PeriodicTimer` cannot hold references to `handler`, `clock` and `interval`.

`java.lang.UnsupportedOperationException` - Thrown if the timer functionality cannot be supported using the `clock` associated with `start` or the given `clock`.

10.4.2 Methods

```
public javax.realtime.ReleaseParameters116
    createReleaseParameters()
```

Create a release parameters object with new objects containing copies of the values corresponding to this timer. When the `PeriodicTimer` interval is greater than 0, create a `PeriodicParameters122` object with a start time and period that correspond to the next firing (or skipping) time, and interval, of this timer. When the interval is 0, create an `AperiodicParameters129` object, since in this case the timer behaves like a `OneShotTimer369`.

If this timer is active, then the start time is the next firing (or skipping) time returned as an `AbsoluteTime320`. Otherwise, the start time is the initial firing (or skipping) time, as set by the last call to `reschedule367`, or if there was no such call, by the constructor of this timer.

Overrides: `createReleaseParameters363` in class `Timer354`

Returns: A new release parameters object with new objects containing copies of the values corresponding to this timer. If the interval is greater than zero, return a new instance of `PeriodicParameters122`. If the interval is zero return a new instance of `AperiodicParameters129`.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*.

public [javax.realtime.Clock₃₅₂](#) [getClock\(\)](#)

Returns the instance of [Clock₃₅₂](#) that the `PeriodicTimer` interval is associated with at the time of the call, even when the time value of interval is zero and the `PeriodicTimer` firing behavior is that of a [OneShotTimer₃₆₉](#).

Overrides: [getClock₃₆₅](#) in class [Timer₃₅₄](#)

Returns: The instance of [Clock₃₅₂](#) that the interval is associated with at the time of the call.

Throws:

`java.lang.IllegalStateException` - Thrown if this `Timer` has been destroyed.

Since: 1.0.1

public [javax.realtime.AbsoluteTime₃₂₀](#) [getFireTime\(\)](#)

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. If the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. If the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.

Overrides: [getFireTime₃₆₅](#) in class [Timer₃₅₄](#)

Returns: The absolute time at which `this` is next expected to fire or to skip, in a newly allocated [AbsoluteTime₃₂₀](#) object. If the timer has been created or re-scheduled (see [Timer.reschedule\(HighResolutionTime\)₃₆₇](#)) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if it is *not-active*.

```
public javax.realtime.AbsoluteTime320 getFireTime(
    javax.realtime.AbsoluteTime320 dest)
```

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. If the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. If the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.

Overrides: `getFireTime366` in class `Timer354`

Parameters:

`dest` - The instance of `AbsoluteTime320` which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is `null` a new object is allocated for the result.

Returns: The instance of `AbsoluteTime320` passed as parameter, with time values representing the absolute time at which `this` is expected to fire or to skip. If the `dest` parameter is `null` the result is returned in a newly allocated object. If the timer has been created or re-scheduled (see `Timer.reschedule(HighResolutionTime)367`) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

Throws:

`java.lang.ArithmeticException` - Thrown if the result does not fit in the normalized format.

`java.lang.IllegalStateException` - Thrown if this `Timer` has been *destroyed*, or if it is *not-active*.

Since: 1.0.1

```
public javax.realtime.RelativeTime332 getInterval()
```

Gets the interval of this `Timer`.

Returns: The `RelativeTime` instance assigned as this periodic timer's interval by the constructor or `setInterval(RelativeTime)377`.

Throws:

`java.lang.IllegalStateException` - Thrown if this Timer has been *destroyed*.

```
public void setInterval(  
    javax.realtime.RelativeTime332 interval)
```

Reset the interval value of this.

Parameters:

`interval` - A `RelativeTime`₃₃₂ object which is the interval used to reset this Timer. A null interval is interpreted as `RelativeTime(0,0)`.

The interval does not affect the first firing (or skipping) of a timer's activation. At each firing (or skipping), the next fire (or skip) time of an *active* periodic timer is established based on the interval currently in use. Resetting the interval of an *active* periodic timer only effects future fire (or skip) times after the next.

Throws:

`java.lang.IllegalArgumentException` - Thrown if interval is a `RelativeTime` instance with a value less than zero.

`IllegalAssignmentError`₄₄₈ - Thrown if this `PeriodicTimer` cannot hold a reference to interval.

`java.lang.IllegalStateException` - Thrown if this Timer has been *destroyed*.

Unofficial

Chapter 11

Asynchrony

This section contains classes that manage asynchrony. They:

- Provide mechanisms that bind the execution of program logic to the occurrence of internal and external events.
- Provide mechanisms that allow asynchronous transfer of control.
- Provide mechanisms that facilitate the asynchronous termination of real-time threads.

Definitions

The following terms and abbreviations will be used:

AE - Asynchronous Event. An instance of the `javax.realtime.AsyncEvent` class.

AEH - Asynchronous Event Handler. An instance of the `javax.realtime.AsyncEventHandler` class.

Bound AEH - Bound Asynchronous Event Handler. An instance of the `javax.realtime.BoundAsyncEventHandler` class.

ATC - Asynchronous Transfer of Control.

AIE - Asynchronously Interrupted Exception. An instance of `javax.realtime.AsynchronouslyInterruptedException` class (a subclass of `java.lang.InterruptedException`).

AI-method - Asynchronously Interruptible method. A method or constructor that includes `AsynchronouslyInterruptedException` explicitly (that is not a subclass of `AsynchronouslyInterruptedException`) in its throws clause.

A *happening* is an event that takes place outside the Java runtime environment. The triggers for happenings depend on the external environment, but happenings might include signals and interrupts.

Lexical Scope [of a method, constructor, or statement]. The textual region within the constructor, method, or statement, excluding the code within any class declarations, and the code within any class instance creation expressions for anonymous classes, contained therein. The lexical scope of a construct does not include the bodies of any methods or constructors that this code invokes.

ATC-deferred section. A synchronized statement, a static initializer or any method or constructor without `AsynchronouslyInterruptedException` in its throws clause. As specified in the introduction to Chapter 8 in *Java Language Specification*, a synchronized method is equivalent to a non-synchronized method with the body of the method contained in a synchronized statement. Thus, a synchronized AI method behaves like an AI method containing only an ATC-deferred statement.

Interruptible blocking methods. The RTSJ and standard Java methods that are explicitly interruptible by an AIE. The interruptible blocking methods comprise `HighResolutionTime.waitForObject()`, `Object.wait()`, `Thread.sleep()`, `RealtimeThread.sleep()`, `Thread.join()`, `ScopedMemory.join()`, `ScopedMemory.joinAndEnter()`, `RealtimeThread.waitForNextPeriodInterruptible()`, `WaitFreeWriteQueue.read()`, `WaitFreeReadQueue.waitForData()`, `WaitFreeReadQueue.write()`, `WaitFreeDequeue.blockingRead()`, `WaitFreeDequeue.blockingWrite()` and their overloaded forms.

Overview

This specification provides several facilities for arranging asynchronous control of execution. These facilities fall into two main categories: asynchronous event handling and asynchronous transfer of control, which includes real-time thread termination.

Asynchronous event handling is captured by the classes `AsyncEvent` (AE), `AsyncEventHandler` (AEH) and `BoundAsyncEventHandler`. An AE is an object used to direct event occurrences to asynchronous event handlers. An event occurrence may be initiated by application logic, by mechanisms internal to the RTSJ implementation (see the handlers in `PeriodicParameters`), or by the triggering of a *happening* external to the JVM (such as a software signal or a hardware interrupt

handler). An event occurrence is initiated in program logic by the invocation of the `fire()` method of an AE. The triggering of an event due to a happening is implementation dependent except as specified in `POSIXSignalHandler`,

An AEH is a schedulable object embodying code that is released for execution in response to the occurrence of an associated event. Each AEH behaves as if it is executed by a `RealtimeThread` or `NoHeapRealtimeThread` except that it is not permitted to use the `waitForNextPeriod()` or `waitForNextPeriodInterruptible()` methods, and it is treated as having a null thread group in all cases. There is not necessarily a separate real-time thread for each AEH, but the server real-time thread (returned by `currentRealtimeThread()`) remains constant during each execution of the `run()` method. The class `BoundAsyncEventHandler` extends `AsyncEventHandler` and ensures that a handler has a dedicated server real-time thread (a server thread is associated with one and only one bound AEH for the lifetime of that AEH). An event count (called `fireCount`) is maintained so that a handler can cope with event bursts - situations where an event occurs more frequently than its handler can respond.

The `interrupt()` method in `java.lang.Thread` provides rudimentary asynchronous communication by setting a pollable/resettable flag in the target thread, and by throwing a synchronous exception when the target thread is blocked at an invocation of `wait()`, `sleep()`, or `join()`. This specification extends the effect of `Thread.interrupt()` by adding an overridden version in `RealtimeThread`, offering a more comprehensive and non-polling asynchronous execution control facility. It is based on throwing and propagating exceptions that, though asynchronous, are deferred where necessary in order to avoid data structure corruption. The main elements of ATC are embodied in the class `AsynchronouslyInterruptedException`, its subclass `Timed`, the interface `Interruptible`, and in the semantics of the `interrupt` method in `RealtimeThread`.

A method indicates its eligibility to be asynchronously interrupted by including the checked exception `AsynchronouslyInterruptedException` in its `throws` clause. If a schedulable object is asynchronously interrupted while executing such a method, then an AIE will be delivered as soon as the schedulable object is outside of a section in which ATC is deferred. Several idioms are available for handling an AIE, giving the programmer the choice of using `catch` clauses and a low-level mechanism with specific control over propagation, or a higher-level facility that allows specifying the interruptible code, the handler, and the result retrieval as separate methods.

Semantics and Requirements for Asynchronous Events and their Handlers

This following list establishes the semantics and requirements that are applicable to asynchronous events and their handlers. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. When an asynchronous event occurs (by either program logic or by the triggering of a happening), its attached handlers (that is, AEHs that have been added to the AE by the execution of `addHandler()`) are released for execution. Every occurrence of an event increments the `fireCount` in each attached handler. Handlers may elect to execute logic for each occurrence of the event or not.
2. The release of attached handlers occurs in execution eligibility order (priority order with the default `PriorityScheduler`) and at the active priority of the schedulable object that invoked the `fire` method. The release of handlers resulting from a happening or a timer must begin within a bounded time (ignoring time consumed by unrelated activities in the system). This worst-case response interval must be documented for some reference architecture.
3. The release of attached handlers is an atomic operation with respect to adding and removing handlers.
4. The logical release of an attached handler may occur before the previous release has completed.
5. A deadline may be associated with each logical release of an attached handler. The deadline is relative to the occurrence of the associated event.
6. AEs and AEHs may be created and used by any program logic within the constraints of the memory assignment rules.
7. More than one AEH may be added to an AE. However, adding an AEH to an AE has no effect if the AEH is already attached to the AE.
8. The same AEH may be added to more than one AE.
9. More than one happening may be associated with the same AE, however, binding a happening to an AE has no effect if it is already attached to the AE.
10. By default all AEHs are considered to be daemons (the daemon status being set by their constructors). An AEH can be set to have a non daemon status after it has been created and before it has been attached to an AE.
11. The object returned by `currentRealtimeThread()` while an AEH is running shall behave with respect to memory access and assignment rules as if it were allocated in the same memory area as the AEH.

An RTSJ program terminates when and only when

- all non-daemon threads (either regular Java threads or real-time threads) are terminated, and
- the `fireCounts` of all non-daemon Bound AEHs or non-daemon AEHs are zero and all releases are completed, and
- there are no non-daemon Bound AEHs or AEHs attached to timers or async events associated with happenings.

Semantics and Requirements for Asynchronous Transfer of Control

Asynchronously interrupting a schedulable object consists of the following activities.

- **Generation** of an asynchronous interrupt exception - this is the event in the underlying system that makes the AIE available to the program.
- **Delivery** of the asynchronous interrupt exception to the target schedulable object - this is the action that invokes the search for and execution of an appropriate handler.

Between the generation and delivery, the asynchronous interrupt exception is held *pending*. After delivery, the AIE remains pending until it is **cleared** by the program logic using `clear()`, `happened()` or `doInterruptible()`.

This following list establishes the semantics and requirements that are applicable to ATC. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. An AIE is generated for a given schedulable object, when the `fire()` method is called on an AIE for which the schedulable object is executing within the `doInterruptible()` method, or the `RealtimeThread.interrupt()` method is called; the latter is effectively called when an AIE is generated by internal virtual machine mechanisms (such as an interrupt I/O protocol) that are asynchronous to the execution of program logic which is the target of the AIE. A generated AIE becomes pending upon generation and remains pending until explicitly cleared or replaced by another AIE.
2. The `RealtimeThread.interrupt()` method throws the generic AIE at the target real-time thread and has the behaviors defined for `Thread.interrupt()`. This is the only interaction between the ATC mechanism and the conventional `interrupt()` mechanism.
3. An AIE is delivered to a schedulable object when it is executing in an AI-method

except as indicated below.

4. The generation of an AIE through the `fire()` mechanism behaves as if it set an asynchronously-interrupted status in the schedulable object. If the schedulable object is blocked within an interruptible blocking method, or invokes an interruptible blocking method, when this asynchronously-interrupted status is set, then the invocation immediately completes by throwing the pending AIE and clearing the asynchronously-interrupted status. When a pending AIE is explicitly cleared then the asynchronously-interrupted status is also cleared.
5. Methods which block through mechanisms other than the interruptible blocking methods, (for example, blocking methods in `java.io.*`) must be prevented from blocking indefinitely when invoked from a method with `AsynchronouslyInterruptedException` in its `throws` clause. When an AIE is generated and the target schedulable object's control is blocked inside one of these methods invoked from an AI-method, the implementation may either unblock the blocked call, raise an `InterruptedException` on behalf of the call, or allow the call to complete normally if the implementation determines that the call would eventually unblock.
6. If an AI-method is attempting to acquire an object lock when an associated AIE is generated, the attempt to acquire the lock is abandoned.
7. If control is in the lexical scope of an ATC-deferred section when an AIE (targeted at the executing schedulable object) is generated, the AIE is not delivered until the first subsequent attempt to transfer control to code that is not ATC-deferred. At that point, control is transferred to the `catch` or `finally` clause of the nearest dynamically-enclosing a `try` statement that has a handler for the generated AIE's (that is a handler naming the AIE's class or any of its superclasses, or a `finally` clause) and which is in an ATC-deferred section. Intervening handlers and `finally` clauses that are not in ATC-deferred sections are not executed, but object locks are released. See section 11.3 of *The Java Language Specification* second edition for an explanation of the terms, *dynamically enclosing* and *handler*. The RTSJ uses those JLS definitions unaltered. Note, if synchronized code is abandoned as a result of this control transfer, the associated locks are released.
8. Constructors are allowed to include `AsynchronouslyInterruptedException` in their `throws` clause and if they do will be asynchronously interruptible under the same conditions as AI methods.
9. Native methods that include `AsynchronouslyInterruptedException` in their `throws` clause have implementation-specific behavior.
10. An implementation must deliver the transfer of control in a schedulable object that is subject to asynchronous interruption (in an AI-method but not in a syn-

chronized block) within a bounded execution time of that schedulable object. This worst-case response interval must be documented for some reference architecture.

11. Instances of the `Timed` class logically have an associated timer. When the timer fires, the schedulable object executing the instance's `doInterruptible` method must have the AIE generated within a bounded execution time of the schedulable object. This worst-case response interval must be documented for some reference architecture.
12. An AIE only has the semantics defined here if it originates with the `AsynchronouslyInterruptedException.fire()` method, the `RealtimeThread.interrupt()` method or from within the real-time VM. If an AIE is thrown from program logic using the Java `throw` statement, it acts the same as throwing any other instance of a subclass of `Exception`, it is processed as a normal exception, and has no affect on the pending state of any AIE, and no affect on the firing of the AIE concerned.

Summary of ATC Operation

The RTSJ's approach to ATC is designed to follow the above principles. It is based on exceptions and is an extension of the current Java language rules for `java.lang.Thread.interrupt()`. In summary, ATC works as follows:

If `so` is an instance of a schedulable object and the `interrupt()` method is called on the real-time thread associated with that object (in this context, the associated real-time thread of an AEH is the real-time thread returned by a call of the `RealtimeThread.currentRealtimeThread()` method by that AEH) then:

1. If control is in an ATC-deferred section, then the AIE remains in a pending state.
2. If control is not in an ATC-deferred section, then control is transferred to the catch or `finally` clause of the nearest dynamically-enclosing a `try` statement that has a handler for the generated AIE's (that is a handler naming the AIE's class or any of its superclasses, or a `finally` clause) and which is in an ATC-deferred section. Intervening handlers and `finally` clauses that are not in ATC-deferred sections are not executed, but objects locks are released. See section 11.3 of *The Java Language Specification* second edition for an explanation of the terms, *dynamically enclosing* and *handles*. The RTSJ uses those definitions unaltered.
3. If control is in an interruptible blocking method the schedulable object is awakened and the generated AIE (which is a subclass of `InterruptedException`) is thrown with regular Java semantics (the AIE is still marked as pending). Then ATC follows option 1, or 2 as appropriate.
4. If control is in an ATC-deferred section, control continues normally until the first attempt to return to an AI method or invoke an AI method or exit a synchronized

block within an AI method. Then ATC follows option 1, or 2 as appropriate.

5. If control is transferred from an ATC-deferred section to an AI method through the action of propagating an exception and if an AIE is pending then when the transition to the AI-method occurs, the thrown exception is discarded and replaced by the AIE.

An AIE may be generated while another AIE is pending. Because AI code blocks are nested by method invocation (a stack-based nesting) there is a natural precedence among active instances of AIE. Let AIE_0 be the AIE raised when the `RealtimeThread.interrupt()` method is invoked and AIE_i ($i = 1, \dots, n$, for n unique instances of AIE) be the AIE generated when `AIE.fire()` is invoked. In the following, the phrase “a frame deeper on the stack than this frame” refers to a method nearer to the current stack frame. The phrase “a frame shallower on the stack than this frame” refers to a method further from the current stack frame.

1. If the current AIE is an AIE_0 and the new AIE is an AIE_x associated with any frame on the stack then the new AIE (AIE_x) is discarded.
2. If the current AIE is an AIE_x and the new AIE is an AIE_0 , then the current AIE (AIE_x) is replaced by the new AIE (AIE_0).
3. If the current AIE is an AIE_x and the new AIE is an AIE_y from a frame deeper on the stack, then the new AIE (AIE_y) discarded.
4. If the current AIE is an AIE_x and the new AIE is an AIE_y from a frame shallower on the stack, the current AIE (AIE_x) is replaced by the new AIE (AIE_y).
5. If the current AIE is an AIE_0 and the new AIE is an AIE_0 , or if the current AIE is an AIE_x and the new AIE is an AIE_x , the new AIE is discarded.

When `clear()` or `happened()` is called on a pending AIE or that AIE is superseded by another, the first AIE’s pending state is cleared. If the `happened()` method is called on a non-pending AIE the result depends on the value of the `propagate` parameter, as indicated in the “No Match” column of the table below. Clearing a non-pending AIE (with the `clear()` method) has no effect.

	Match	No Match
<code>propagate == true</code>	clear the pending AIE, return true	the AIE remains pending, propagate
<code>propagate == false</code>	clear the pending AIE, return true	the AIE remains pending, return false

Rationale

The design of the asynchronous event handling facilities was intended to provide the necessary functionality while allowing efficient implementations and catering for a variety of real-time applications. In particular, in some real-time systems there may be a large number of potential events and event handlers (numbering in the thousands or perhaps even the tens of thousands), although at any given time only a small number will be used. Thus it would not be appropriate to dedicate a real-time thread to each event handler. The RTSJ addresses this issue by allowing the programmer to specify an event handler either as not bound to a specific real-time thread (the class `AsyncEventHandler`) or alternatively as bound to a dedicated real-time thread (the class `BoundAsyncEventHandler`). The RTSJ does not define at what point a non-bound event handler is bound to a real-time thread for its execution.

Events are dataless: the `fire` method does not pass any data to the handler. This was intentional in the interest of simplicity and efficiency. An application that needs to associate data with an `AsyncEvent` can do so explicitly by setting up a buffer; it will then need to deal with buffer overflow issues as required by the application.

The ability to trigger an ATC in a schedulable object is necessary in many kinds of real-time applications but must be designed carefully in order to minimize the risks of problems such as data structure corruption and deadlock. There is, invariably, a tension between the desire to cause an ATC to be immediate, and the desire to ensure that certain sections of code are executed to completion.

One basic decision was to allow ATC in a method only if the method explicitly permits this. The default of no ATC is reasonable, since legacy code might be written expecting no ATC, and asynchronously aborting the execution of such a method could lead to unpredictable results. Since the natural way to model ATC is with an exception (`AsynchronouslyInterruptedException`), the way that a method indicates its susceptibility to ATC is by including `AsynchronouslyInterruptedException` in its `throws` clause. Causing this exception to be thrown in a real-time thread `t` as an effect of calling `t.interrupt()` was a natural extension of the semantics of `interrupt` as currently defined by `java.lang.Thread`.

One ATC-deferred section is `synchronized` code. This is a context that needs to be executed completely in order to ensure a program operates correctly. If `synchronized` code were aborted, a shared object could be left in an inconsistent state. Note that by making `synchronized` code ATC-deferred, this specification avoids the problems that caused `Thread.stop()` to be deprecated and that have made the use of `Thread.destroy()`, (now also deprecated in Java 1.5) prone to deadlock. If `synchronized` code calls an AI-method and an associated AIE is generated, then if no appropriate handler is present in the `synchronized` code, the AIE will propagate through the code.

Constructors and `finally` clauses are subject to interruption if the program indicates so. However, if a constructor is aborted, an object might be only partially initialized. If the execution of a `finally` clause in an AI-method is aborted, needed cleanup code might not be performed. Indeed, a `finally` clause in an aborted AI-method will not be executed at all if the abort occurs before its execution begins. It is the programmer's responsibility to ensure that executing these constructs either does not induce unwanted ATC latency (if ATCs are not allowed) or does not produce undesirable results (if ATCs are allowed).

A potential problem with using the exception mechanism to model ATC is that a method with a "catch-all" handler (for example a `catch` clause identifying `Exception` or even `Throwable` as the exception class) can inadvertently intercept an exception intended for a caller. This problem is avoided by having special semantics for catching an AIE. Even though a `catch` clause may catch an AIE, the exception will be propagated unless the handler invokes the `happened` method from AIE. Thus, if a schedulable object is asynchronously interrupted while in a try block that has a handler such as

```
catch (Throwable e){ return; }
```

the AIE will remain pending and will be thrown next time control enters or returns to an AI method.

This specification does not provide a special mechanism for terminating a real-time thread; ATC can be used to achieve this effect. This means that, by default, a real-time thread cannot be asynchronously terminated; to support asynchronous termination it needs to enter methods that are AI enabled at frequent intervals. Allowing termination as the default would have been questionable, bringing the same insecurities that are found in `Thread.stop()` and `Thread.destroy()`.

11.1 AsyncEvent

Declaration

```
public class AsyncEvent
```

Direct Known Subclasses: [Timer](#)₃₅₄

Description

An asynchronous event can have a set of handlers associated with it, and when the event occurs, the `fireCount` of each handler is incremented, and the handlers are released (see [AsyncEventHandler](#)₃₉₃).

11.1.1 Constructors

```
public AsyncEvent()
```

Create a new AsyncEvent object.

11.1.2 Methods

```
public void addHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Add a handler to the set of handlers associated with this event. An instance of AsyncEvent may have more than one associated handler. However, adding a handler to an event has no effect if the handler is already attached to the event.

The execution of this method is atomic with respect to the execution of the fire() method.

Since this affects the constraints expressed in the release parameters of an existing schedulable object, this may change the feasibility of the current system. This method does not change feasibility set of any scheduler, and no feasibility test is performed.

Note, there is an implicit reference to the handler stored in this. The assignment must be valid under any applicable memory assignment rules.

Parameters:

handler - The new handler to add to the list of handlers already associated with this.

If the handler is already associated with the event, the call has no effect.

Throws:

java.lang.IllegalArgumentException - Thrown if handler is null.

IllegalAssignmentError₄₄₈ - Thrown if this AsyncEvent cannot hold a reference to handler.

```
public void bindTo(java.lang.String happening)
```

Binds this to an external event, a *happening*. The meaningful values of happening are implementation dependent. This instance of AsyncEvent is considered to have occurred whenever the happening is triggered. More

than one happening can be bound to the same `AsyncEvent`. However, binding a happening to an event has no effect if the happening is already bound to the event.

When an event, which is declared in a scoped memory area, is bound to an external happening, the reference count of that scoped memory area is incremented (as if there is an external real-time thread accessing the area). The reference count is decremented when the event is unbound from the happening.

Parameters:

`happening` - An implementation dependent value that binds this instance of `AsyncEvent` to a happening.

Throws:

`UnknownHappeningException457` - Thrown if the String value is not supported by the implementation.

`java.lang.IllegalArgumentException` - Thrown if `happening` is null.

```
public javax.realtime.ReleaseParameters116
    createReleaseParameters()
```

Create a `ReleaseParameters116` object appropriate to the release characteristics of this event. The default is the most pessimistic:

`AperiodicParameters129`. This is typically called by code that is setting up a handler for this event that will fill in the parts of the release parameters for which it has values, e.g., cost. The returned `ReleaseParameters116` object is not bound to the event. Any changes in the event's release parameters are not reflected in previously returned objects.

If an event returns `PeriodicParameters122`, there is no requirement for an implementation to check that the handler is released periodically.

Returns: A new `ReleaseParameters116` object.

```
public void fire()
```

Fire this instance of `AsyncEvent`. The asynchronous event handlers associated with this event will be released. If no handlers are attached then the method does nothing. An `AsyncEvent` that has been bound to an external happening can still be fired by the application code.

- If the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters object of type `AperiodicParameters` attached and the execution of `AsyncEvent.fire()`

introduces the requirement to throw at least one type of exception, then all instances of `AsyncEventHandler` not affected by the exception are handled normally.

- If the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters object of type `SporadicParameters` attached and the execution of `AsyncEvent.fire()` introduces the simultaneous requirement to throw more than one type of exception or error then `MITViolationException453` has precedence over `ArrivalTimeQueueOverflowException446`.

Throws:

`MITViolationException453` - Thrown under the base priority scheduler's semantics if there is a handler associated with this event that has its MIT violated by the call to fire (and it has set the minimum inter-arrival time violation behavior to `MITViolationExcept`). Only the handlers which do not have their MITs violated are released in this situation.

`ArrivalTimeQueueOverflowException446` - Thrown if the queue of arrival time information overflows. Only the handlers which do not cause this exception to be thrown are released in this situation.

```
public boolean handledBy(
    javax.realtime.AsyncEventHandler393 handler)
```

Test to see if the handler given as the parameter is associated with this.

Parameters:

handler - The handler to be tested to determine if it is associated with this.

Returns: True if the parameter is associated with this. False if handler is null or the parameters is not associated with this.

```
public void removeHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Remove a handler from the set associated with this event. The execution of this method is atomic with respect to the execution of the `fire()` method.

A removed handler continues to execute until its `fireCount` becomes zero and it completes.

Since this affects the constraints expressed in the release parameters of an existing schedulable object, this may change the feasibility of the current

system. This method does not change the feasibility set of any scheduler, and no feasibility test is performed.

Parameters:

`handler` - The handler to be disassociated from `this`. If null nothing happens. If the `handler` is not already associated with `this` then nothing happens.

```
public void setHandler(
    javax.realtime.AsyncEventHandler393 handler)
```

Associate a new handler with this event and remove all existing handlers. The execution of this method is atomic with respect to the execution of the `fire()` method.

Since this affects the constraints expressed in the release parameters of the existing schedulable objects, this may change the feasibility of the current system. This method does not change the feasibility set of any scheduler, and no feasibility test is performed.

Parameters:

`handler` - The new instance of `AsyncEventHandler393` to be associated with `this`. If `handler` is null then no handler will be associated with `this` (i.e., remove all handlers).

Throws:

`IllegalAssignmentError448` - Thrown if this `AsyncEvent` cannot hold a reference to `handler`.

```
public void unbindTo(java.lang.String happening)
```

Removes a binding to an external event, a *happening*. The meaningful values of `happening` are implementation dependent. If the associated event is declared in a scoped memory area, the reference count for the memory area is decremented.

Parameters:

`happening` - An implementation dependent value representing some external event to which this instance of `AsyncEvent` is bound.

Throws:

`UnknownHappeningException457` - Thrown if this instance of `AsyncEvent` is not bound to the given `happening` or the given `happening` is not supported by the implementation.

`java.lang.IllegalArgumentException` - Thrown if `happening` is null.

11.2 AsyncEventHandler

Declaration

```
public class AsyncEventHandler implements Schedulable81
```

All Implemented Interfaces: `java.lang.Runnable`, [Schedulable₈₁](#)

Direct Known Subclasses: [BoundAsyncEventHandler₄₁₈](#)

Description

An asynchronous event handler encapsulates code that is released after an instance of [AsyncEvent₃₈₈](#) to which it is attached occurs.

It is guaranteed that multiple releases of an event handler will be serialized. It is also guaranteed that (unless the handler explicitly chooses otherwise) for each release of the handler, there will be one execution of the [handleAsyncEvent\(\)₄₀₂](#) method. Control over the number of calls to [handleAsyncEvent\(\)₄₀₂](#) is given by methods which manipulate a `fireCount`. These may be called by the application via subclassing and overriding [handleAsyncEvent\(\)₄₀₂](#).

Instances of `AsyncEventHandler` with a release parameter of type [SporadicParameters₁₃₆](#) or [AperiodicParameters₁₂₉](#) have a list of release times which correspond to the occurrence times of instances of [AsyncEvent₃₈₈](#) to which they are attached. The minimum interarrival time specified in [SporadicParameters₁₃₆](#) is enforced when a release time is added to the list. Unless the handler explicitly chooses otherwise, there will be one execution of the code in [handleAsyncEvent\(\)₄₀₂](#) for each entry in the list.

The deadline and the time each release event causes the AEH to become eligible for execution are properties of the scheduler that controls the AEH. For the base scheduler, the deadline for each release event is relative to its fire time, and the release takes place at fire time but execution eligibility may be deferred if the queue's MIT violation policy is SAVE.

Handlers may do almost anything a real-time thread can do. They may run for a long or short time, and they may block. (Note: blocked handlers may hold system resources.) A handler may not use the [RealtimeThread.waitForNextPeriod\(\)₅₄](#) method.

Normally, handlers are bound to an execution context dynamically when the instances of [AsyncEvent₃₈₈](#)s to which they are bound occur. This can introduce a (small) time penalty. For critical handlers that can not afford the expense, and where this penalty is a problem, [BoundAsyncEventHandler₄₁₈](#)s can be used.

The scheduler for an asynchronous event handler is inherited from the thread/schedulable object that created it. If it was created from a Java thread, the scheduler is the current default scheduler.

The semantics for memory areas that were defined for real-time threads apply in the same way to instances of `AsyncEventHandler`. They may inherit a scope stack when they are created, and the single parent rule applies to the use of memory scopes for instances of `AsyncEventHandler` just as it does in real-time threads.

11.2.1 Constructors

`public AsyncEventHandler()`

Create an instance of `AsyncEventHandler` with default values for all parameters. This constructor is equivalent to `AsyncEventHandler(null, null, null, null, null, false, null)`.

`public AsyncEventHandler(boolean nonheap)`

Create an instance of `AsyncEventHandler` with the specified `nonheap` flag and default values for all other parameters. This constructor is equivalent to `AsyncEventHandler(null, null, null, null, null, nonheap, null)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if this is in heap memory, and `nonheap` is true.

`public AsyncEventHandler(boolean nonheap, java.lang.Runnable logic)`

Create an instance of `AsyncEventHandler` with the specified `nonheap` flag and `logic` value, and default values for all other parameters. This constructor is equivalent to `AsyncEventHandler(null, null, null, null, null, nonheap, logic)`.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `logic` or this is in heap memory, and `nonheap` is true.

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to `logic`.

```
public AsyncEventHandler(java.lang.Runnable logic)
```

Create an instance of `AsyncEventHandler` with the given `logic` parameter and default values for all other parameters. This constructor is equivalent to `AsyncEventHandler(null, null, null, null, null, false, logic)`.

Throws:

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to `logic`.

```
public AsyncEventHandler(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.MemoryArea161 area,
    javax.realtime.ProcessingGroupParameters143 group,
    boolean nonheap)
```

Create an instance of `AsyncEventHandler` with the specified values for scheduling parameters, release parameters, memory parameters, initial memory area, processing group parameters and non-heap flag. This constructor is equivalent to: `AsyncEventHandler(scheduling, release, memory, area, group, nonheap, null)`

Throws:

`java.lang.IllegalArgumentException` - Thrown if `nonheap` is true and this or any object passed as a parameter is in heap memory. Also thrown if `nonheap` is true and `area` is heap memory.

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `scheduling`, `release`, `memory` and `group`, or if those parameters cannot hold a reference to the new `AsyncEventHandler`. Also thrown if the new `AsyncEventHandler` instance cannot hold a reference to `area`.

```
public AsyncEventHandler(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
```

```

javax.realtime.MemoryArea161 area,
javax.realtime.ProcessingGroupParameters143 group,
boolean nonheap,
java.lang.Runnable logic)

```

Create an instance of `AsyncEventHandler` with the specified parameters.

Parameters:

`scheduling` - A [SchedulingParameters₁₁₂](#) object which will be associated with the constructed instance. If null, and the creator is a Java thread, a `SchedulingParameters` object is created which has the default scheduling parameters value for the scheduler associated with the current thread. If null, and the creator is a schedulable object, the scheduling parameters are inherited from the current schedulable object (a new `SchedulingParameters` object is cloned).

`release` - A [ReleaseParameters₁₁₆](#) object which will be associated with the constructed instance. If null, this will have default `ReleaseParameters` for this AEH's scheduler.

`memory` - A [MemoryParameters₂₇₃](#) object which will be associated with the constructed instance. If null, this will have no `MemoryParameters`.

`area` - The [MemoryArea₁₆₁](#) for this. If null, the memory area will be that of the current thread/schedulable object.

`group` - A [ProcessingGroupParameters₁₄₃](#) object which will be associated with the constructed instance. If null, this will not be associated with any processing group.

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a [NoHeapRealtimeThread₅₅](#). A false value means this will have characteristics identical to a [RealtimeThread₂₉](#). If true and the current thread/schedulable object is *not* executing within a [ScopedMemory₁₇₂](#) or [ImmortalMemory₁₆₈](#) scope then an `java.lang.IllegalArgumentException` is thrown.

`logic` - The `java.lang.Runnable` object whose `run()` method is executed by [handleAsyncEvent\(\)₄₀₂](#). If null, the default [handleAsyncEvent\(\)₄₀₂](#) method invokes nothing.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `nonheap` is true and any parameter, or `this` is in heap memory or `area` is heap memory.

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `scheduling`, `release`, `memory`, and `group`, or if those parameters cannot hold a reference to the new `AsyncEventHandler`. Also thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `area` and `logic`.

```
public AsyncEventHandler(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.MemoryArea161 area,
    javax.realtime.ProcessingGroupParameters143 group,
    java.lang.Runnable logic)
```

Create an instance of `AsyncEventHandler` with the specified values for scheduling parameters, release parameters, memory parameters, initial memory area, processing group parameters and runnable logic. This constructor is equivalent to: `AsyncEventHandler(scheduling, release, memory, area, group, false, logic)`

Throws:

`java.lang.IllegalArgumentException` - Thrown if `nonheap` is true and `logic`, any parameter object, or `this` is in heap memory. Also thrown if `noheap` is true and `area` is heap memory.

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `scheduling`, `release`, `memory`, and `group`, or if those parameters cannot hold a reference to the new `AsyncEventHandler`. Also thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `area` and `logic`.

11.2.2 Methods

`public boolean addIfFeasible()`

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis with `this` added to the system. If the resulting system is feasible, inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should be considered in feasibility analysis until further notified. If the analysis showed that the system including `this` would not be feasible, this method does not admit `this` to the feasibility set.

If the object is already included in the feasibility set, do nothing.

Specified By: [addIfFeasible₈₁](#) in interface [Schedulable₈₁](#)

Returns: True if inclusion of `this` in the feasibility set yields a feasible system, and false otherwise. If true is returned then `this` is known to be in the feasibility set. If false is returned `this` was not added to the feasibility set, but may already have been present.

`public boolean addToFeasibility()`

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should be considered in feasibility analysis until further notified.

If the object is already included in the feasibility set, do nothing.

Specified By: [addToFeasibility₈₂](#) in interface [Schedulable₈₁](#)

Returns: True, if the resulting system is feasible. False, if not.

`protected int getAndClearPendingFireCount()`

This is an accessor method for `fireCount`. This method atomically sets the value of `fireCount` to zero and returns the value from before it was set to zero. This may be used by handlers for which the logic can accommodate multiple releases in a single execution. The general form for using this is:

```

public void handleAsyncEvent() {
    int numberOfReleases = getAndClearPendingFireCount();
    <handle the events>
}

```

The effect of a call to `getAndClearPendingFireCount` on the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Returns: The value held by `fireCount` prior to setting the value to zero.

protected int getAndDecrementPendingFireCount()

This is an accessor method for `fireCount`. This method atomically decrements, by one, the value of `fireCount` (if it was greater than zero) and returns the value from before the decrement. This method can be used in the `handleAsyncEvent()` method to handle multiple releases:

```

public void handleAsyncEvent() {
    <setup>
    do {
        <handle the event>
    } while(getAndDecrementPendingFireCount(>0));
}

```

This construction is necessary only in the case where a handler wishes to avoid the setup costs since the framework guarantees that `handleAsyncEvent()` will be invoked whenever the `fireCount` is greater than zero. The effect of a call to `getAndDecrementPendingFireCount` on the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Returns: The value held by `fireCount` prior to decrementing it by one.

protected int getAndIncrementPendingFireCount()

This is an accessor method for `fireCount`. This method atomically increments, by one, the value of `fireCount` and returns the value from before the increment.

The effect of a call to `getAndIncrementPendingFireCount` on the arrival-time queue and the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Returns: The value held by `fireCount` prior to incrementing it by one.

Throws:

MITViolationException₄₅₃ - Thrown when this AEH is controlled by sporadic scheduling parameters under the base

scheduler, the parameters specify the `mitViolationExcept` policy, and this method would introduce a release that would violate the specified minimum interarrival time.

[ArrivalTimeQueueOverflowException](#)₄₄₆ - Thrown when this AEH is controlled by aperiodic scheduling parameters under the base scheduler, the release parameters specify the `arrivalTimeQueueOverflowExcept` policy, and this method would cause the arrival time queue to overflow.

```
public javax.realtime.MemoryArea161 getMemoryArea()
```

This is an accessor method for the initial instance of [MemoryArea](#)₁₆₁ associated with `this`.

Returns: The instance of [MemoryArea](#)₁₆₁ which was passed as the area parameter when `this` was created (or the default value if area was allowed to default. To determine the current status of the memory area stack associated with `this`, use the static methods defined in the [RealtimeThread](#)₂₉ class. That is [RealtimeThread.getCurrentMemoryArea\(\)](#)₃₃, [RealtimeThread.getInitialMemoryAreaIndex\(\)](#)₃₃, [RealtimeThread.getMemoryAreaStackSize\(\)](#)₃₄.

```
public javax.realtime.MemoryParameters273
    getMemoryParameters()
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁
Gets a reference to the [MemoryParameters](#)₂₇₃ object for this schedulable object.

Specified By: [getMemoryParameters](#)₈₂ in interface [Schedulable](#)₈₁

Returns: A reference to the current [MemoryParameters](#)₂₇₃ object.

```
protected int getPendingFireCount()
```

This is an accessor method for `fireCount`. The `fireCount` field nominally holds the number of times associated instances of [AsyncEvent](#)₃₈₈ have occurred that have not had the method `handleAsyncEvent()` invoked. It is incremented and decremented by the implementation of the RTSJ. The application logic may manipulate the value in this field for application-specific reasons.

Returns: The value held by `fireCount`.

**public [javax.realtime.ProcessingGroupParameters₁₄₃](#)
[getProcessingGroupParameters\(\)](#)**

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [ProcessingGroupParameters₁₄₃](#) object for this schedulable object.

Specified By: [getProcessingGroupParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [ProcessingGroupParameters₁₄₃](#) object.

**public [javax.realtime.ReleaseParameters₁₁₆](#)
[getReleaseParameters\(\)](#)**

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [ReleaseParameters₁₁₆](#) object for this schedulable object.

Specified By: [getReleaseParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [ReleaseParameters₁₁₆](#) object.

public [javax.realtime.Scheduler₉₇](#) [getScheduler\(\)](#)

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [Scheduler₉₇](#) object for this schedulable object.

Specified By: [getScheduler₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the associated [Scheduler₉₇](#) object.

**public [javax.realtime.SchedulingParameters₁₁₂](#)
[getSchedulingParameters\(\)](#)**

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Gets a reference to the [SchedulingParameters₁₁₂](#) object for this schedulable object.

Specified By: [getSchedulingParameters₈₂](#) in interface [Schedulable₈₁](#)

Returns: A reference to the current [SchedulingParameters₁₁₂](#) object.

public void handleAsyncEvent()

This method holds the logic which is to be executed when any [AsyncEvent₃₈₈](#) with which this handler is associated is fired. This method will be invoked repeatedly while `fireCount` is greater than zero.

The default implementation of this method invokes the `run` method of any non-null `Logic` instance passed to the constructor of this handler.

If the initial memory area of this `AsyncEventHandler` is a [ScopedMemory₁₇₂](#), the initial memory area's reference count does not drop below one between invocations of `handleAsyncEvent()` unless this `AsyncEventHandler` becomes *unfireable*.

All throwables from (or propagated through) `handleAsyncEvent` are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

public final boolean isDaemon()

Tests if this event handler is a daemon handler.

Returns: True if this event handler is a daemon handler; false otherwise.

Since: 1.0.1

public boolean removeFromFeasibility()

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Inform the scheduler and cooperating facilities that this instance of [Schedulable₈₁](#) should *not* be considered in feasibility analysis until it is further notified.

Specified By: [removeFromFeasibility₈₃](#) in interface [Schedulable₈₁](#)

Returns: True, if the removal was successful. False, if the schedulable object cannot be removed from the scheduler's feasibility set; e.g., the schedulable object is not part of the scheduler's feasibility set.

public final void run()

When used as part of the internal mechanism activated by firing an async event, this method's detailed semantics are defined by the scheduler associated with this handler. The general outline is:

```

while (fireCount > 0) {
    [initiate release]
    fireCount--;
    try {
        handleAsyncEvent();
    } catch (Throwable th){
        th.printStackTrace();
    }
    [effect completion]
}

```

All throwables from (or propagated through) `handleAsyncEvent()`⁴⁰² are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

When it is directly invoked, this method invokes `handleAsyncEvent()`⁴⁰² repeatedly while the `fireCount` is greater than zero; e.g.,

```

while (getAndDecrementPendingFireCount() > 0)
    handleAsyncEvent();

```

however direct invocation of `run` is not recommended as it may interact with the normal release of this handler.

Applications cannot override this method and thus should use the `logic` parameter at construction, or override `handleAsyncEvent()` in subclasses with the logic of the handler.

Specified By: `run` in interface `Runnable`

public final void setDaemon(boolean on)

Marks this event handler as either a daemon event handler or a user event handler. The Real-Time Virtual Machine exits when the only schedulable objects and threads running are all daemon. This method must be called before the event handler is attached to any event. Once attached, it cannot be changed.

Parameters:

`on` - If true, marks this event handler as a daemon handler.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if this event handler is attached to an AE.

`java.lang.SecurityException` - Thrown if the current schedulable object cannot modify this event handler.

Since: 1.0.1

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of `this`. If the resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setIfFeasible₈₃](#) in interface [Schedulable₈₁](#)

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if `this` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is

currently waiting for the next release in
[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or
[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setIfFeasible](#)₈₄ in interface [Schedulable](#)₈₁

Parameters:

release - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

memory - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

group - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃ .)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable

object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or

[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

```
public boolean setIfFeasible(
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setIfFeasible](#)₈₅ in interface [Schedulable](#)₈₁

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made.
False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `RealtimeThread.waitForNextPeriod()`₅₄ or `RealtimeThread.waitForNextPeriodInterruptible()`₅₄.

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: `javax.realtime.Schedulable`₈₁

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the resulting system is feasible, this method replaces the current parameters of this with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setIfFeasible`₈₆ in interface `Schedulable`₈₁

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in [RealtimeThread.waitForNextPeriod\(\)₅₄](#) or [RealtimeThread.waitForNextPeriodInterruptible\(\)₅₄](#).

```
public boolean setIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis using the proposed parameter objects as replacements for the current parameters of this. If the

resulting system is feasible, this method replaces the current parameters of `this` with the proposed ones.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setIfFeasible`₈₈ in interface `Schedulable`₈₁

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

`group` - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter values are not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and any of the proposed parameter objects are located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if `this` cannot hold references to the proposed parameter objects, or the parameter objects cannot hold a reference to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `RealtimeThread.waitForNextPeriod()`⁵⁴ or `RealtimeThread.waitForNextPeriodInterruptible()`⁵⁴.

```
public void setMemoryParameters(
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: `javax.realtime.Schedulable`⁸¹
Sets the memory parameters associated with this instance of `Schedulable`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the memory parameters of the existing schedulable objects, this may change the feasibility of the current system.

Specified By: `setMemoryParameters`⁸⁹ in interface `Schedulable`⁸¹

Parameters:

`memory` - A `MemoryParameters`²⁷³ object which will become the memory parameters associated with this after the method call. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`¹⁰³.)

Throws:

`java.lang.IllegalArgumentException` - Thrown if `memory` is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and `memory` is located in heap memory.

`IllegalAssignmentError`⁴⁴⁸ - Thrown if the schedulable object cannot hold a reference to `memory`, or if `memory` cannot hold a reference to this schedulable object instance.

```
public boolean setMemoryParametersIfFeasible(
    javax.realtime.MemoryParameters273 memory)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setMemoryParametersIfFeasible₉₀](#) in interface [Schedulable₈₁](#)

Parameters:

`memory` - The proposed memory parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError448` - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

```
public void setProcessingGroupParameters(
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Sets the [ProcessingGroupParameters₁₄₃](#) of this.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be

immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Since this affects the constraints expressed in the processing group parameters of the existing schedulable objects, this may change the feasibility of the current system.

Specified By: `setProcessingGroupParameters90` in interface `Schedulable81`

Parameters:

`group` - A `ProcessingGroupParameters143` object which will take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler103`.)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `group` is not compatible with the scheduler for this schedulable object. Also thrown if this schedulable object is no-heap and `group` is located in heap memory.

`IllegalAssignmentError448` - Thrown if this object cannot hold a reference to `group` or `group` cannot hold a reference to this.

```
public boolean setProcessingGroupParametersIfFeasible(
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: `javax.realtime.Schedulable81`

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of this. If the resulting system is feasible, this method replaces the current parameter of this with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setProcessingGroupParametersIfFeasible91` in interface `Schedulable81`

Parameters:

group - The proposed processing group parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError₄₄₈](#) - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

```
public void setReleaseParameters(
    javax.realtime.ReleaseParameters116 release)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)

Sets the release parameters associated with this instance of `Schedulable`.

Since this affects the constraints expressed in the release parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. The different properties of the release parameters may take effect at different times. See the documentation for the scheduler for details.

Specified By: [setReleaseParameters₉₂](#) in interface [Schedulable₈₁](#)

Parameters:

release - A [ReleaseParameters₁₁₆](#) object which will become the release parameters associated with this after the method call, and take effect as determined by the associated scheduler. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `release` is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and `release` is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if `this` object cannot hold a reference to `release` or `release` cannot hold a reference to `this`.

`java.lang.IllegalThreadStateException` - Thrown if the new `release` parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in `RealtimeThread.waitForNextPeriod()`₅₄ or `RealtimeThread.waitForNextPeriodInterruptible()`₅₄.

```
public boolean setReleaseParametersIfFeasible(
    javax.realtime.ReleaseParameters116 release)
```

Description copied from interface: `javax.realtime.Schedulable`₈₁

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. If the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: `setReleaseParametersIfFeasible`₉₃ in interface `Schedulable`₈₁

Parameters:

`release` - The proposed release parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See `PriorityScheduler`₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if this cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

`RealtimeThread.waitForNextPeriod()`₅₄ or

`RealtimeThread.waitForNextPeriodInterruptible()`₅₄.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler)
```

Description copied from interface: `javax.realtime.Schedulable`₈₁

Sets the reference to the Scheduler object. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and scheduler.

Specified By: `setScheduler`₉₄ in interface `Schedulable`₈₁

Parameters:

`scheduler` - A reference to the scheduler that will manage execution of this schedulable object. Null is not a permissible value.

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduler` is null, or the schedulable object's existing parameter values are not compatible with `scheduler`. Also thrown if this schedulable object is no-heap and `scheduler` is located in heap memory.

`IllegalAssignmentError`₄₄₈ - Thrown if the schedulable object cannot hold a reference to `scheduler`.

`java.lang.SecurityException` - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setScheduler(
    javax.realtime.Scheduler97 scheduler,
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memoryParameters,
    javax.realtime.ProcessingGroupParameters143 group)
```

Description copied from interface: [javax.realtime.Schedulable₈₁](#)
 Sets the scheduler and associated parameter objects. The timing of the change must be agreed between the scheduler currently associated with this schedulable object, and scheduler.

Specified By: [setScheduler₉₄](#) in interface [Schedulable₈₁](#)

Parameters:

- `scheduler` - A reference to the scheduler that will manage the execution of this schedulable object. Null is not a permissible value.
- `scheduling` - A reference to the [SchedulingParameters₁₁₂](#) which will be associated with this. If null, the default value is governed by `scheduler` (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `release` - A reference to the [ReleaseParameters₁₁₆](#) which will be associated with this. If null, the default value is governed by `scheduler` (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `memoryParameters` - A reference to the [MemoryParameters₂₇₃](#) which will be associated with this. If null, the default value is governed by `scheduler` (a new object is created if the default value is not null). (See [PriorityScheduler₁₀₃](#).)
- `group` - A reference to the [ProcessingGroupParameters₁₄₃](#) which will be associated with this. If null, the default value is governed by `scheduler` (a new object is created). (See [PriorityScheduler₁₀₃](#).)

Throws:

- `java.lang.IllegalArgumentException` - Thrown when `scheduler` is null or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable object is no-heap and `scheduler`, `scheduling`, `release`, `memoryParameters`, or `group` is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if this object cannot hold references to all the parameter objects or the parameters cannot hold references to this.

`java.lang.IllegalThreadStateException` - Thrown if the new release parameters change the schedulable object from periodic scheduling to some other protocol and the schedulable object is currently waiting for the next release in

[RealtimeThread.waitForNextPeriod\(\)](#)₅₄ or

[RealtimeThread.waitForNextPeriodInterruptible\(\)](#)₅₄.

`java.lang.SecurityException` - Thrown if the caller is not permitted to set the scheduler for this schedulable object.

```
public void setSchedulingParameters(
    javax.realtime.SchedulingParameters112 scheduling)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

Sets the scheduling parameters associated with this instance of `Schedulable`.

Since this affects the scheduling parameters of the existing schedulable objects, this may change the feasibility of the current system.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

Specified By: [setSchedulingParameters](#)₉₅ in interface [Schedulable](#)₈₁

Parameters:

`scheduling` - A reference to the [SchedulingParameters](#)₁₁₂ object. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Throws:

`java.lang.IllegalArgumentException` - Thrown when `scheduling` is not compatible with the associated scheduler. Also thrown if this schedulable object is no-heap and `scheduling` is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if this object cannot hold a reference to `scheduling` or `scheduling` cannot hold a reference to this.

```
public boolean setSchedulingParametersIfFeasible(
    javax.realtime.SchedulingParameters112 scheduling)
```

Description copied from interface: [javax.realtime.Schedulable](#)₈₁

This method first performs a feasibility analysis using the proposed parameter object as replacement for the current parameter of `this`. If the resulting system is feasible, this method replaces the current parameter of `this` with the proposed one.

This change becomes effective under conditions determined by the scheduler controlling the schedulable object. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

This method does not require that the schedulable object be in the feasibility set before it is called. If it is not initially a member of the feasibility set it will be added if the resulting system is feasible.

Specified By: [setSchedulingParametersIfFeasible](#)₉₆ in interface [Schedulable](#)₈₁

Parameters:

`scheduling` - The proposed scheduling parameters. If null, the default value is governed by the associated scheduler (a new object is created if the default value is not null). (See [PriorityScheduler](#)₁₀₃.)

Returns: True, if the resulting system is feasible and the changes are made. False, if the resulting system is not feasible and no changes are made.

Throws:

`java.lang.IllegalArgumentException` - Thrown when the parameter value is not compatible with the schedulable object's scheduler. Also thrown if this schedulable object is no-heap and the proposed parameter object is located in heap memory.

[IllegalAssignmentError](#)₄₄₈ - Thrown if `this` cannot hold a reference to the proposed parameter object, or the parameter object cannot hold a reference to `this`.

11.3 BoundAsyncEventHandler

Declaration

```
public class BoundAsyncEventHandler extends
    AsyncEventHandler393
```

All Implemented Interfaces: `java.lang.Runnable`, [Schedulable](#)₈₁

Description

A bound asynchronous event handler is an instance of [AsyncEventHandler](#)₃₉₃ that is permanently bound to a dedicated real-time thread. Bound asynchronous event handlers are for use in situations where the added timeliness is worth the overhead of dedicating an individual real-time thread to the handler. Individual server real-time threads can only be dedicated to a single bound event handler.

11.3.1 Constructors

```
public BoundAsyncEventHandler()
```

Create an instance of `BoundAsyncEventHandler` using default values. This constructor is equivalent to `BoundAsyncEventHandler(null, null, null, null, null, false, null)`

```
public BoundAsyncEventHandler(
    javax.realtime.SchedulingParameters112 scheduling,
    javax.realtime.ReleaseParameters116 release,
    javax.realtime.MemoryParameters273 memory,
    javax.realtime.MemoryArea161 area,
    javax.realtime.ProcessingGroupParameters143 group,
    boolean nonheap,
    java.lang.Runnable logic)
```

Create an instance of `BoundAsyncEventHandler` with the specified parameters.

Parameters:

`scheduling` - A [SchedulingParameters](#)₁₁₂ object which will be associated with the constructed instance. If null, and the creator is a Java thread, a `SchedulingParameters` object is created which has the default `SchedulingParameters` for the scheduler associated with the current thread. If null, and the creator is a schedulable object, the `SchedulingParameters` are inherited from the current schedulable object (a new `SchedulingParameters` object is cloned).

`release` - A [ReleaseParameters](#)₁₁₆ object which will be associated with the constructed instance. If null, this will have default `ReleaseParameters` for the BAEH's scheduler.

`memory` - A [MemoryParameters₂₇₃](#) object which will be associated with the constructed instance. If null, this will have no `MemoryParameters`.

`area` - The [MemoryArea₁₆₁](#) for this. If null, the memory area will be that of the current thread/schedulable object.

`group` - A [ProcessingGroupParameters₁₄₃](#) object which will be associated with the constructed instance. If null, this will not be associated with any processing group.

`logic` - The `java.lang.Runnable` object whose `run()` method is executed by [AsyncEventHandler.handleAsyncEvent\(\)₄₀₂](#). If null, the default [AsyncEventHandler.handleAsyncEvent\(\)₄₀₂](#) method invokes nothing.

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a [NoHeapRealtimeThread₅₅](#). A false value means this will have characteristics identical to a [RealtimeThread₂₉](#). If true and the current thread/schedulable object is *not* executing within a [ScopedMemory₁₇₂](#) or [ImmortalMemory₁₆₈](#) scope then an `java.lang.IllegalArgumentException` is thrown.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `nonheap` is true and `logic`, any parameter object, or `this` is in heap memory. Also thrown if `noheap` is true and `area` is heap memory.

`IllegalAssignmentError448` - Thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `scheduling`, `release`, `memory` and `group`, or if those parameters cannot hold a reference to the new `AsyncEventHandler`. Also thrown if the new `AsyncEventHandler` instance cannot hold a reference to non-null values of `area` and `logic`.

11.4 Interruptible

Declaration

```
public interface Interruptible
```

Description

`Interruptible` is an interface implemented by classes that will be used as arguments on the method `doInterruptible()` of [AsynchronouslyInterruptedException₄₂₂](#) and its subclasses. `doInterruptible()` invokes the implementation of the method in this interface.

11.4.1 Methods

```
public void interruptAction(
    javax.realtime.AsynchronouslyInterruptedException422
    exception)
```

This method is called by the system if the `run()` method is interrupted. Using this, the program logic can determine if the `run()` method completed normally or had its control asynchronously transferred to its caller.

Parameters:

`exception` - The currently pending AIE. Used to invoke methods on [AsynchronouslyInterruptedException₄₂₂](#) from within the `interruptAction()` method.

```
public void run(
    javax.realtime.AsynchronouslyInterruptedException422
    exception)
    throws AsynchronouslyInterruptedException
```

The main piece of code that is executed when an implementation is given to `doInterruptible()`. When a class is created that implements this interface (for example through an anonymous inner class) it must include the `throws` clause to make the method interruptible. If the `throws` clause is omitted the `run()` method will not be interruptible.

Parameters:

`exception` - The AIE object whose `doInterruptible` method is calling the `run` method. Used to invoke methods on [AsynchronouslyInterruptedException₄₂₂](#) from within the `run()` method.

Throws:

[AsynchronouslyInterruptedException₄₂₂](#)

11.5 AsynchronouslyInterruptedException

Declaration

```
public class AsynchronouslyInterruptedException extends  
    java.lang.InterruptedException
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `Timed428`

Description

A special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a schedulable object.

A schedulable object that is executing a method or constructor, which is declared with an `AsynchronouslyInterruptedException422` in its throws clause, can be asynchronously interrupted except when it is executing in the lexical scope of a synchronized statement within that method/constructor. As soon as the schedulable object leaves the lexical scope of the method by calling another method/constructor it may be asynchronously interrupted if the called method/constructor is asynchronously interruptible. (See this chapter's introduction section for the detailed semantics).

The asynchronous interrupt is generated for a real-time thread, `t`, when the `t.interrupt()` method is called or the `fire` method is called of an AIE for which `t` has a `doInterruptible` method call in progress.

The interrupt is generated for an AEH (or BAEH), `h`, if the `fire` method is called of an AIE for which `h` has a `doInterruptible` method call in progress.

If an asynchronous interrupt is generated when the target real-time thread/schedulable object is executing within an ATC-deferred section, the asynchronous interrupt becomes pending. A pending asynchronous interrupt is delivered when the target real-time thread/schedulable object next attempts to enter asynchronously interruptible code.

Asynchronous transfers of control (ATCs) are intended to allow long-running computations to be terminated without the overhead or latency of polling with `java.lang.Thread.interrupted()`.

When `RealtimeThread.interrupt()36`, or `AsynchronouslyInterruptedException.fire()` is called, the `AsynchronouslyInterruptedException` is compared against any currently pending `AsynchronouslyInterruptedException` on the schedulable object. If there is none,

or if the depth of the `AsynchronouslyInterruptedException` is less than the currently pending `AsynchronouslyInterruptedException`; (i.e., it is targeted at a less deeply nested method call), the new `AsynchronouslyInterruptedException` becomes the currently pending `AsynchronouslyInterruptedException` and the previously pending `AsynchronouslyInterruptedException` is discarded. Otherwise, the new `AsynchronouslyInterruptedException` is discarded.

When an `AsynchronouslyInterruptedException` is caught, the catch clause may invoke the `clear()` method on the `AsynchronouslyInterruptedException` in which it is interested to see if the exception matches the pending `AsynchronouslyInterruptedException`. If so, the pending `AsynchronouslyInterruptedException` is cleared for the schedulable object and `clear` returns true. Otherwise, the current AIE remains pending and `clear` returns false.

`RealtimeThread.interrupt()` generates a system-wide generic `AsynchronouslyInterruptedException` which will always propagate outward through interruptible methods until the generic `AsynchronouslyInterruptedException` is identified and handled. The pending state of the generic AIE is per-schedulable object.

Other sources (e.g., `AsynchronouslyInterruptedException.fire()` and `Timed428`) will generate specific instances of `AsynchronouslyInterruptedException` which applications can identify and thus limit propagation.

`AsyncEventHandler393` objects should interact with the ATC mechanisms via the `Interruptible420` interface.

11.5.1 Constructors

```
public AsynchronouslyInterruptedException()
```

Create an instance of `AsynchronouslyInterruptedException`.

11.5.2 Methods

```
public boolean clear()
```

Atomically see if this is pending on the currently executing schedulable object, and if so, make it non-pending.

Returns: True if this was pending. False if this was not pending.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if called from a Java thread.

Since: 1.0.1

`public boolean disable()`

Disable the throwing of this exception. If the `fire` method is called on this AIE whilst it is disabled, the fire is held pending and delivered as soon as the AIE is enabled and the interruptible code is within an AI-method. If an AIE is pending when the associated `disable` method is called, the AIE remains pending, and is delivered as soon as the AIE is enabled and the interruptible code is within an AI-method.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `disable` returns `false` and does nothing.

Note: disabling the genericAIE associated with a real-time thread only affects the firing of that AIE. If the genericAIE is generated by the `RealtimeThread.interrupt()`₃₆ mechanism, the AIE is delivered (unless the Interruptible code is in an AI-deferred region, in which case it is marked as pending and handled in the usual way).

Returns: True if this was enabled before the method was called and the call was invoked with the associated `doInterruptible()` in progress. False: otherwise.

`public boolean doInterruptible(javax.realtime.Interruptible420 logic)`

Executes the `run()` method of the given `Interruptible420`. This method may be on the stack in exactly one `Schedulable81` object. An attempt to invoke this method in a schedulable object while it is on the stack of another or the same schedulable object will cause an immediate return with a value of `false`.

The `run` method of given `Interruptible` is always entered with the exception in the enabled state, but that state can be modified with `enable()`₄₂₅ and `disable()`₄₂₄ and the state can be observed with `isEnabled()`₄₂₇.

The `interruptAction` method of the given `Interruptible420` is called if any AIE is generated for the schedulable object executing the `doInterruptible` method. The generated AIE remains pending after the

`interruptAction` method has finished if the pending AIE is not the AIE executing the `doInterruptible` method. If it is, the pending AIE is cleared.

Parameters:

`logic` - An instance of an `Interruptible`₄₂₀ whose `run()` method will be called.

Returns: True if the method call completed normally. False if another call to `doInterruptible` has not completed.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if called from a Java thread.

`java.lang.IllegalArgumentException` - Thrown if a null parameter was passed.

public boolean enable()

Enable the throwing of this exception. This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `enable` returns false and does nothing.

Returns: True if this was disabled before the method was called and the call was invoked whilst the associated `doInterruptible()` is in progress. False: otherwise.

public boolean fire()

Generate this exception if its `doInterruptible()` has been invoked and not completed. If this is the only outstanding AIE on the schedulable object that invoked this AIE's `doInterruptible(Interruptible)`₄₂₄ method, this AIE becomes that schedulable object's current AIE. Otherwise, it only becomes the current AIE if it is at a less deeper level of nesting compared with the current outstanding AIE.

Returns: True if this is not disabled and it has an invocation of a `doInterruptible()` in progress and there is no outstanding fire request. False otherwise.

public static

`javax.realtime.AsynchronouslyInterruptedException422`
getGeneric()

Gets the singleton system generic `AsynchronouslyInterruptedException` that is generated when `RealtimeThread.interrupt()36` is invoked.

Returns: The generic `AsynchronouslyInterruptedException`.

Throws:

`java.lang.IllegalThreadStateException` - if the current thread is a Java thread.

public boolean **happened**(boolean propagate)

Deprecated. Since 1.0.1. This method seriously violates standard Java exception semantics, and while it is a convenience it is not required. The `happened` method can be replaced with the `clear` method and application logic.

Used with an instance of this exception to see if the current exception is this exception. When an `AsynchronouslyInterruptedException` is caught, the catch clause may invoke the `happened()` method on the `AsynchronouslyInterruptedException` in which it is interested to see if it matches the pending `AsynchronouslyInterruptedException`. If so, the pending `AsynchronouslyInterruptedException` is cleared for the schedulable object and `happened` returns true. Otherwise, the behavior of `happened` depends on its propagation parameter. If propagation parameter is true, the `AsynchronouslyInterruptedException` will continue to propagate outward; i.e., it will be re-thrown by a mechanism that bypassed the normal requirement that the checked exception be identified in the method's signature. If propagation parameter is false, `happened` will return false and the `AsynchronouslyInterruptedException` remains pending.

Parameters:

propagate - Control the behavior when this is not the current exception:

If true and this exception is the current one, set the state of this to non pending and return true.

If true and this exception is not the current one, propagate the exception; i.e., rethrow it.

If false and this exception is the current one, the state of this is set to nonpending (i.e., it will stop propagating) and return true.

If false and this exception is not the current one, return false.

Returns: True if this is the current exception. False if this is not the current exception.

Throws:

`java.lang.IllegalThreadStateException` - Thrown if called from a Java thread.

`public boolean isEnabled()`

Query the enabled status of this exception.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `isEnabled` returns false and does nothing.

Returns: True if this is enabled and the method call was invoked in the context of the associated `doInterruptible()`. False otherwise.

`public static void propagate()`

Deprecated. Since 1.0.1. This method seriously violates standard Java exception semantics, and while it is a convenience it is not required. It should be replaced with `throw` of an instance of `AsynchronouslyInterruptedException`.

Cause the pending exception to continue up the stack. The current AIE remains pending and control is transferred immediately to the next suitable catch or finally clause under the normal rules for AIE propagation.

If there is no current AIE, the method does nothing and simply returns.

This method is normally used in a catch clause that is handling an AIE, but that is not required. The method may be invoked at any time (from a schedulable object).

Throws:

`java.lang.IllegalThreadStateException` - Thrown if called from a Java thread.

11.6 Timed

Declaration

```
public class Timed extends AsynchronouslyInterruptedException422
```

All Implemented Interfaces: `java.io.Serializable`

Description

Create a scope in a [Schedulable](#)₈₁ object which will be asynchronously interrupted at the expiration of a timer. This timer will begin measuring time at some point between the time `doInterruptible()` is invoked and the time the `run()` method of the `Interruptible` object is invoked. Each call of `doInterruptible()` on an instance of `Timed` will restart the timer for the amount of time given in the constructor or the most recent invocation of `resetTime()`. The timer is cancelled if it has not expired before the `doInterruptible()` method has finished.

All memory use of an instance of `Timed` occurs during construction or the first invocation of `doInterruptible()`. Subsequent invocations of `doInterruptible()` do not allocate memory.

If the timer fires, the resulting AIE will be generated for the schedulable object within a bounded execution time of the targeted schedulable object.

Typical usage: `new Timed(T).doInterruptible(interruptible);`

11.6.1 Constructors

```
public Timed(javax.realtime.HighResolutionTime314 time)
```

Create an instance of `Timed` with a timer set to `time`. If the `time` is in the past the [AsynchronouslyInterruptedException](#)₄₂₂ mechanism is activated immediately after or when the `doInterruptible()` method is called.

Parameters:

`time` - If `time` is a [RelativeTime](#)₃₃₂ value, it is the interval of time between the invocation of `doInterruptible()` and when the schedulable object is asynchronously interrupted. If `time` is an [AbsoluteTime](#)₃₂₀ value, the timer asynchronously interrupts at this time (assuming the timer has not been cancelled).

Throws:

`java.lang.IllegalArgumentException` - Thrown if `time` is null.

11.6.2 Methods

```
public boolean doInterruptible(  
    javax.realtime.Interruptible420 logic)
```

Execute a time-out method. Starts the timer and executes the run() method of the given `Interruptible420` object.

Overrides: `doInterruptible424` in class `AsynchronouslyInterruptedException422`

Parameters:

logic - Implements an `Interruptible420` run() method.

Throws:

`java.lang.IllegalArgumentException` - Thrown if logic is null.

```
public void resetTime(  
    javax.realtime.HighResolutionTime314 time)
```

To set the time-out for the next invocation of `doInterruptible()`.

Parameters:

time - This can be an absolute time or a relative time. If null the time-out is not changed.

Unofficial

Chapter 12

System and Options

This section contains classes describe the handling of POSIX signals, or related to the system as a whole. These classes:

- Provide a common idiom for binding POSIX signals to instances of `AsyncEventHandler` when POSIX signals are available on the underlying platform.
- Provide a class that contains operations and semantics that affect the entire system.
- Provide the security semantics required by the additional features in the entirety of this specification, which are additional to those required by implementations of the Java Language Specification.

The `RealtimeSecurity` class provides security primarily for physical memory access.

Semantics and Requirements

This list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. The POSIX signal handler class is required to be available when implementations of this specification execute on an underlying platform that provides POSIX signals or any subset of signals named with the POSIX names.

2. The `RealtimeSecurity` class is required.
3. If applications execute the method call, `System.getProperty("javax.realtime.version")`, the return value will be a string of the form, "x.y.z". Where 'x' is the major version number and 'y' and 'z' are minor version numbers. These version numbers state to which version of the RTSJ the underlying implementation claims conformance. The first release of the RTSJ, dated 11/2001, is numbered as, 1.0.0. Since this property is required in only subsequent releases of the RTSJ implementations of the RTSJ which intend to conform to 1.0.0 may return the String "1.0.0" or null.

Rationale

This specification accommodates the variation in underlying system variation in a number of ways. One of the most important is the concept of optionally required classes (e.g., the POSIX signal handler class). This class provides a commonality that can be relied upon by program logic that intends to execute on implementations that themselves execute on POSIX compliant systems.

The `RealtimeSystem` class functions in similar capacity to `java.lang.System`. Similarly, the `RealtimeSecurity` class functions similarly to `java.lang.SecurityManager`.

12.1 POSIXSignalHandler

Declaration

```
public final class POSIXSignalHandler
```

Description

Use instances of `AsyncEventHandler393` to handle POSIX signals. Usage:

```
POSIXSignalHandler.addHandler(SIGINT, intHandler);
```

This class is required to be implemented only if the underlying operating system supports POSIX signals.

POSIX requires the implementation to support 13 non-real-time signals and it names 12 additional optional signals. This class mirrors that practice. Except for the 13 signals required by POSIX, support for a signal in this class does not imply that it is generated by the runtime system.

12.1.1 Fields

`public static final int SIGABRT`

Used by abort.

`public static final int SIGALRM`

Alarm clock.

`public static final int SIGBUS`

Bus error.

This is an optional POSIX signal.

`public static final int SIGCANCEL`

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Thread cancellation signal used by libthread.

`public static final int SIGCHLD`

Child status change alias.

`public static final int SIGCLD`

Child status change.

This is an optional POSIX signal.

`public static final int SIGCONT`

Stopped process has been continued.

This is an optional POSIX signal.

`public static final int SIGEMT`

EMT instruction.

This is an optional POSIX signal.

public static final int SIGFPE

Floating point exception.

public static final int SIGFREEZE

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Special signal used by CPR.

public static final int SIGHUP

Hang-up.

public static final int SIGILL

Illegal instruction (not reset when caught).

public static final int SIGINT

Interrupt (rubout).

public static final int SIGIO

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Socket I/O possible (SIGPOLL alias).

public static final int SIGIOT

IOT instruction.

This is an optional POSIX signal.

public static final int SIGKILL

Kill (cannot be caught or ignored).

public static final int SIGLOST

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Resource lost (e.g., record-lock lost).

public static final int SIGLWP

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Special signal used by thread library.

public static final int SIGPIPE

Write on a pipe with no one to read it.

public static final int SIGPOLL

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Pollable event occurred.

public static final int SIGPROF

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Profiling timer expired.

public static final int SIGPWR

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Power-fail restart.

public static final int SIGQUIT

Quit (ASCII FS).

public static final int SIGSEGV

Segmentation violation.

public static final int SIGSTOP

Stop (cannot be caught or ignored).

This is an optional POSIX signal.

public static final int SIGSYS

Bad argument to system call.
This is an optional POSIX signal.

public static final int SIGTERM

Software termination signal from kill.

public static final int SIGTHAW

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.
Special signal used by CPR.

public static final int SIGTRAP

Trace trap.
This is an optional POSIX signal.

public static final int SIGTSTP

User stop requested from TTY.
This is an optional POSIX signal.

public static final int SIGTTIN

Background tty read attempted.
This is an optional POSIX signal.

public static final int SIGTTOU

Background tty write attempted.
This is an optional POSIX signal.

public static final int SIGURG

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.
Urgent socket condition.

public static final int SIGUSR1

User defined signal 1.

public static final int SIGUSR2

User defined signal 2.

public static final int SIGVTALRM

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Virtual timer expired.

public static final int SIGWAITING

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Process's light-weight processes are blocked.

public static final int SIGWINCH

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Window size change.

public static final int SIGXCPU

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Exceeded cpu limit.

public static final int SIGXFSZ

Deprecated. 1.0.1 This signal is not mentioned in the POSIX 9945-1-1996 standard.

Exceeded file size limit.

12.1.2 Methods

```
public static void addHandler(
    int signal,
    javax.realtime.AsyncEventHandler393 handler)
```

Add the given [AsyncEventHandler393](#) to the list of handlers of the given signal.

Parameters:

`signal` - One of the POSIX signals from this (e.g., `this.SIGINT`). If the value given to `signal` is not one of the those defined by this class then an `java.lang.IllegalArgumentException` will be thrown.

`handler` - An [AsyncEventHandler393](#) which will be scheduled when the given signal occurs.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `signal` is not known by this class or if `handler` is null.

`java.lang.UnsupportedOperationException` - Thrown if `signal` is known to this class, but not supported on this implementation.

```
public static void removeHandler(
    int signal,
    javax.realtime.AsyncEventHandler393 handler)
```

Remove the given [AsyncEventHandler393](#) from the list of handlers of the given signal.

Parameters:

`signal` - One of the POSIX signals from this (e.g., `this.SIGINT`). If the value given to `signal` is not one of the POSIX signals then an `java.lang.IllegalArgumentException` will be thrown.

`handler` - An [AsyncEventHandler393](#) which will be scheduled when the given signal occurs.

Throws:

`java.lang.IllegalArgumentException` - Thrown if `signal` is not known by this class or if `handler` is null.

```
public static void setHandler(
    int signal,
    javax.realtime.AsyncEventHandler393 handler)
```

Set the given [AsyncEventHandler₃₉₃](#) as the handler of the given signal.

Parameters:

`signal` - One of the POSIX signals from this (e.g., `this.SIGLOST`). If the value given to `signal` is not one of the POSIX signals then an `java.lang.IllegalArgumentException` will be thrown.

`handler` - An [AsyncEventHandler₃₉₃](#) which will be scheduled when the given signal occurs. If `h` is null then no handler will be associated with this (i.e., remove all handlers).

Throws:

`java.lang.IllegalArgumentException` - Thrown if `signal` is not known by this class.

`java.lang.UnsupportedOperationException` - Thrown if `signal` is known to this class, but not supported on this implementation.

12.2 RealtimeSecurity

Declaration

```
public class RealtimeSecurity
```

Description

Security policy object for real-time specific issues. Primarily used to control access to physical memory.

Security requirements are generally application-specific. Every implementation shall have a default `RealtimeSecurity` instance, and a way to install a replacement at run-time, [RealtimeSystem.setSecurityManager\(RealtimeSecurity\)₄₄₄](#).

The default security is minimal. All security managers should prevent access to JVM internal data and the Java heap; additional protection is implementation-specific and must be documented.

12.2.1 Constructors

```
public RealtimeSecurity()
```

Create an `RealtimeSecurity` object.

12.2.2 Methods

```
public void checkAccessPhysical()  
    throws SecurityException
```

Check whether the application is allowed to access physical memory.

Throws:

`java.lang.SecurityException` - The application doesn't have permission to access physical memory.

```
public void checkAccessPhysicalRange(  
    long base,  
    long size)  
    throws SecurityException
```

Checks whether the application is allowed to access physical memory within the specified range.

Parameters:

`base` - The beginning of the address range.

`size` - The size of the address range.

Throws:

`java.lang.SecurityException` - The application doesn't have permission to access the memory in the given range.

```
public void checkAEHSetDaemon()  
    throws SecurityException
```

Checks whether the application is allowed to set the daemon status of an AEH.

Throws:

`java.lang.SecurityException` - Thrown if the application is not permitted to alter the daemon status.

Since: 1.0.1

```
public void checkSetFilter()
    throws SecurityException
```

Checks whether the application is allowed to register [PhysicalMemoryTypeFilter₂₀₅](#) objects with the [PhysicalMemoryManager₁₉₇](#).

Throws:

`java.lang.SecurityException` - The application doesn't have permission to register filter objects.

```
public void checkSetMonitorControl(
    javax.realtime.MonitorControl286 policy)
    throws SecurityException
```

Checks whether the application is allowed to set the default monitor control policy.

Parameters:

`policy` - The new policy

Throws:

`java.lang.SecurityException` - Thrown if the application doesn't have permission to change the default monitor control policy to `policy`.

Since: 1.0.1

```
public void checkSetScheduler()
    throws SecurityException
```

Checks whether the application is allowed to set the scheduler.

Throws:

`java.lang.SecurityException` - The application doesn't have permission to set the scheduler.

12.3 RealtimeSystem

Declaration

```
public final class RealtimeSystem
```

Description

`RealtimeSystem` provides a means for tuning the behavior of the implementation by specifying parameters such as the maximum number of locks that can be in use concurrently, and the monitor control policy. In addition, `RealtimeSystem` provides a mechanism for obtaining access to the security manager, garbage collector and scheduler, to make queries from them or to set parameters.

12.3.1 Fields

`public static final byte BIG_ENDIAN`

Value to indicate the byte ordering for the underlying hardware.

`public static final byte BYTE_ORDER`

The byte ordering of the underlying hardware.

`public static final byte LITTLE_ENDIAN`

Value to indicate the byte ordering for the underlying hardware.

12.3.2 Methods

`public static javax.realtime.GarbageCollector278 currentGC()`

Return a reference to the currently active garbage collector for the heap.

Returns: A [GarbageCollector](#)₂₇₈ object which is the current collector collecting objects on the traditional Java heap.

`public static int getConcurrentLocksUsed()`

Gets the maximum number of locks that have been used concurrently. This value can be used for tuning the concurrent locks parameter, which is used as a hint by systems that use a monitor cache.

Returns: An integer whose value is the number of locks in use at the time of the invocation of the method. If the number of concurrent locks is not tracked by the implementation, return -1. Note that if the number of concurrent locks is not tracked, the number of available concurrent locks is effectively unlimited.

```
public static javax.realtime.MonitorControl286  
    getInitialMonitorControl()
```

Returns the monitor control object that represents the initial monitor control policy.

Returns: The initial monitor control policy.

Since: 1.0.1

```
public static int getMaximumConcurrentLocks()
```

Gets the maximum number of locks that can be used concurrently without incurring an execution time increase as set by the `setMaximumConcurrentLocks()` methods.

Note: Any relationship between this method and `setMaximumConcurrentLocks` is implementation-specific. This method returns the actual maximum number of concurrent locks the platform can currently support, or `Integer.MAX_VALUE` if there is no maximum. The `setMaximumConcurrentLocks` method give the implementation a hint as to the maximum number of concurrent locks it should expect.

Returns: An integer whose value is the maximum number of locks that can be in simultaneous use.

```
public static javax.realtime.RealtimeSecurity439  
    getSecurityManager()
```

Gets a reference to the security manager used to control access to real-time system features such as access to physical memory.

Returns: A [RealtimeSecurity](#)₄₃₉ object representing the default real-time security manager.

```
public static void setMaximumConcurrentLocks(  
    int numLocks)
```

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provide a hint to systems that use a monitor cache as to how much space to dedicate to the cache.

Parameters:

`numLocks` - An integer whose value becomes the number of locks that can be in simultaneous use without incurring an execution time increase. If number is less than or equal to zero nothing happens. If the system does not use this hint this method has no

effect other than on the value returned by `getMaximumConcurrentLocks()`⁴⁴³.

```
public static void setMaximumConcurrentLocks(
    int number,
    boolean hard)
```

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provide a limit for the size of the monitor cache on systems that provide one if `hard` is true.

Parameters:

`number` - The maximum number of locks that can be in simultaneous use without incurring an execution time increase. If `number` is less than or equal to zero nothing happens. If the system does not use this hint this method has no effect other than on the value returned by `getMaximumConcurrentLocks()`⁴⁴³.

`hard` - If true, `number` sets a limit. If a lock is attempted which would cause the number of locks to exceed `number` then a `ResourceLimitError`⁴⁵⁸ is thrown. If the system does not limit use of concurrent locks, this parameter is silently ignored.

```
public static void setSecurityManager(
    javax.realtime.RealtimeSecurity439 manager)
```

Sets a new real-time security manager.

Parameters:

`manager` - A `RealtimeSecurity`⁴³⁹ object which will become the new security manager.

Throws:

`java.lang.SecurityException` - Thrown if security manager has already been set.

Chapter 13

Exceptions

This section contains exceptions defined by the RTSJ. These exception classes:

- Provide additional exception classes required for other sections of this specification.
- Provide the ability to asynchronously transfer the control of program logic (see `AsynchronouslyInterruptedException`.)

Semantics and Requirements

This list establishes the semantics and requirements that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. All classes in this section are required.
2. All exceptions, except `AsynchronouslyInterruptedException`, are required to have semantics exactly as those of their eventual superclass in the `java.*` hierarchy.

The `AsynchronouslyInterruptedException` class is not included in this chapter. It is more closely related to asynchronous operation than to exception handling and so can be found in the Asynchrony chapter.

Rationale

The need for additional exceptions given the new semantics added by the other sections of this specification is obvious. That the specification attaches new, nontraditional, exception semantics to `AsynchronouslyInterruptedException` is, perhaps, not so obvious. However, after careful thought, and given our self-imposed directive that only well-defined code blocks would be subject to having their control asynchronously transferred, the chosen mechanism is logical.

13.1 `ArrivalTimeQueueOverflowException`

Declaration

```
public class ArrivalTimeQueueOverflowException extends
    java.lang.RuntimeException
```

All Implemented Interfaces: `java.io.Serializable`

Description

If an arrival time occurs and should be queued but the queue already holds a number of times equal to the initial queue length defined by this an instance of this class may be thrown. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler` is bound then the arrival time is ignored.

Since: 1.0.1 Becomes unchecked

13.1.1 Constructors

```
public ArrivalTimeQueueOverflowException()
```

A constructor for `ArrivalTimeQueueOverflowException`.

```
public ArrivalTimeQueueOverflowException(
    java.lang.String description)
```

A descriptive constructor for `ArrivalTimeQueueOverflowException`.

Parameters:

`description` - A description of the exception.

13.2 CeilingViolationException

Declaration

```
public class CeilingViolationException extends  
    java.lang.IllegalThreadStateException
```

All Implemented Interfaces: java.io.Serializable

Description

This exception is thrown when a schedulable object or `java.lang.Thread` attempts to lock an object governed by an instance of `PriorityCeilingEmulation288` and the thread or SO's base priority exceeds the policy's ceiling.

13.2.1 Methods

```
public int getCallerPriority()
```

Gets the base priority of the SO or thread whose attempt to synchronize resulted in the throwing of this.

Returns: The synchronizing thread's base priority.

```
public int getCeiling()
```

Gets the ceiling of the `PriorityCeilingEmulation` policy which was exceeded by the base priority of an SO or thread that attempted to synchronize on an object governed by the policy, which resulted in throwing of this.

Returns: The ceiling of the `PriorityCeilingEmulation` policy which caused this exception to be thrown.

13.3 DuplicateFilterException

Declaration

```
public class DuplicateFilterException extends
    java.lang.Exception
```

All Implemented Interfaces: java.io.Serializable

Description

[PhysicalMemoryManager](#)₁₉₇ can only accommodate one filter object for each type of memory. It throws this exception if an attempt is made to register more than one filter for a type of memory.

13.3.1 Constructors

```
public DuplicateFilterException()
```

A constructor for DuplicateFilterException.

```
public DuplicateFilterException(
    java.lang.String description)
```

A descriptive constructor for DuplicateFilterException.

Parameters:

description - Description of the error.

13.4 IllegalAssignmentError

Declaration

```
public class IllegalAssignmentError extends java.lang.Error
```

All Implemented Interfaces: java.io.Serializable

Description

The exception thrown on an attempt to make an illegal assignment. For example, this will be thrown on any attempt to assign a reference to an object in scoped memory (an area of memory identified by an instance of [ScopedMemory](#)₁₇₂) to a field of an object in immortal memory.

13.4.1 Constructors

```
public IllegalAssignmentError()
```

A constructor for IllegalAssignmentError.

```
public IllegalAssignmentError(
    java.lang.String description)
```

A descriptive constructor for IllegalAssignmentError.

Parameters:

description - Description of the error.

13.5 InaccessibleAreaException

Declaration

```
public class InaccessibleAreaException extends
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

The specified memory area is not on the current thread's scope stack.

Since: 1.0.1 Becomes unchecked

13.5.1 Constructors

```
public InaccessibleAreaException()
```

A constructor for InaccessibleAreaException.

```
public InaccessibleAreaException(
    java.lang.String description)
```

A descriptive constructor for InaccessibleAreaException.

Parameters:

description - Description of the error.

13.6 MemoryAccessError

Declaration

```
public class MemoryAccessError extends java.lang.Error
```

All Implemented Interfaces: java.io.Serializable

Description

This error is thrown on an attempt to refer to an object in an inaccessible [MemoryArea₁₆₁](#). For example this will be thrown if logic in a [NoHeapRealtimeThread₅₅](#) attempts to refer to an object in the traditional Java heap.

13.6.1 Constructors

```
public MemoryAccessError()
```

A constructor for MemoryAccessError.

```
public MemoryAccessError(java.lang.String description)
```

A descriptive constructor for MemoryAccessError.

Parameters:

description - Description of the error.

13.7 MemoryInUseException

Declaration

```
public class MemoryInUseException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

There has been attempt to allocate a range of physical or virtual memory that is already in use.

13.7.1 Constructors

```
public MemoryInUseException()
```

A constructor for `MemoryInUseException`.

```
public MemoryInUseException(java.lang.String description)
```

A descriptive constructor for `MemoryInUseException`.

Parameters:

description - Description of the error.

13.8 MemoryScopeException

Declaration

```
public class MemoryScopeException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown if construction of any of the wait-free queues is attempted with the ends of the queue in incompatible memory areas. Also thrown by wait-free queue methods when such an incompatibility is detected after the queue is constructed.

13.8.1 Constructors

```
public MemoryScopeException()
```

A constructor for `MemoryScopeException`.

```
public MemoryScopeException(java.lang.String description)
```

A descriptive constructor for `MemoryScopeException`.

Parameters:

description - A description of the exception.

13.9 MemoryTypeConflictException

Declaration

```
public class MemoryTypeConflictException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

This exception is thrown when the [PhysicalMemoryManager₁₉₇](#) is given conflicting specifications for memory. The conflict can be between types in an array of memory type specifiers, or between the specifiers and a specified base address.

Since: 1.0.1 Changed to an unchecked exception.

13.9.1 Constructors

```
public MemoryTypeConflictException()
```

A constructor for MemoryTypeConflictException.

```
public MemoryTypeConflictException(  
    java.lang.String description)
```

A descriptive constructor for MemoryTypeConflictException.

Parameters:

description - A description of the exception.

13.10 MITViolationException

Declaration

```
public class MITViolationException extends
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown by the [AsyncEvent.fire\(\)₃₉₀](#) or [AsyncEventHandler.getAndIncrementPendingFireCount\(\)₃₉₉](#) on a minimum interarrival time violation. More specifically, it is thrown under the semantics of the base priority scheduler's sporadic parameters' mitViolationExcept policy when an attempt is made to introduce a release that would violate the MIT constraint.

Since: 1.0.1 Becomes unchecked

13.10.1 Constructors

```
public MITViolationException()
```

A constructor for MITViolationException.

```
public MITViolationException(
    java.lang.String description)
```

A descriptive constructor for MITViolationException.

Parameters:

description - Description of the error.

13.11 OffsetOutOfBoundsException

Declaration

```
public class OffsetOutOfBoundsException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown if the constructor of an [ImmutablePhysicalMemory₂₁₂](#), [LTPhysicalMemory₂₂₁](#), [VTPhysicalMemory₂₃₀](#), [RawMemoryAccess₂₃₉](#), or [RawMemoryFloatAccess₂₆₂](#) is given an invalid address.

Since: 1.0.1 Becomes unchecked

13.11.1 Constructors

```
public OffsetOutOfBoundsException()
```

A constructor for `OffsetOutOfBoundsException`.

```
public OffsetOutOfBoundsException(  
    java.lang.String description)
```

A descriptive constructor for `OffsetOutOfBoundsException`.

Parameters:

description - A description of the exception.

13.12 ScopedCycleException

Declaration

```
public class ScopedCycleException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown when a schedulable object attempts to enter an instance of [ScopedMemory₁₇₂](#) where that operation would cause a violation of the single parent rule.

13.12.1 Constructors

```
public ScopedCycleException()
```

A constructor for `ScopedCycleException`.

```
public ScopedCycleException(java.lang.String description)
```

A descriptive constructor for `ScopedCycleException`.

Parameters:

description - Description of the error.

13.13 SizeOutOfBoundsException

Declaration

```
public class SizeOutOfBoundsException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown if the constructor of an [ImmutablePhysicalMemory₂₁₂](#), [LTPhysicalMemory₂₂₁](#), [VTPhysicalMemory₂₃₀](#), [RawMemoryAccess₂₃₉](#), or [RawMemoryFloatAccess₂₆₂](#) is given an invalid size or if an accessor method on one of the above classes would cause access to an invalid address.

Since: 1.0.1 Becomes unchecked

13.13.1 Constructors

```
public SizeOutOfBoundsException()
```

A constructor for `SizeOutOfBoundsException`.

```
public SizeOutOfBoundsException(  
    java.lang.String description)
```

A descriptive constructor for `SizeOutOfBoundsException`.

Parameters:

`description` - The description of the exception.

13.14 **ThrowBoundaryError**

Declaration

```
public class ThrowBoundaryError extends java.lang.Error
```

All Implemented Interfaces: `java.io.Serializable`

Description

The error thrown by `MemoryArea.enter(Runnable)`¹⁶⁴ when a `java.lang.Throwable` allocated from memory that is not usable in the surrounding scope tries to propagate out of the scope of the `enter`.

13.14.1 Constructors

```
public ThrowBoundaryError()
```

A constructor for `ThrowBoundaryError`.

```
public ThrowBoundaryError(java.lang.String description)
```

A descriptive constructor for `ThrowBoundaryError`.

Parameters:

`description` - Description of the error.

13.15 UnsupportedPhysicalMemoryException

Declaration

```
public class UnsupportedPhysicalMemoryException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown when the underlying hardware does not support the type of physical memory requested from an instance of one of the physical memory or raw memory access classes.

Since: 1.0.1 Becomes unchecked

See Also: [RawMemoryAccess₂₃₉](#), [RawMemoryFloatAccess₂₆₂](#),
[ImmortalPhysicalMemory₂₁₂](#), [LTPhysicalMemory₂₂₁](#),
[VTPhysicalMemory₂₃₀](#)

13.15.1 Constructors

```
public UnsupportedPhysicalMemoryException()
```

A constructor for `UnsupportedPhysicalMemoryException`.

```
public UnsupportedPhysicalMemoryException(  
    java.lang.String description)
```

A descriptive constructor for `UnsupportedPhysicalMemoryException`.

Parameters:

description - The description of the exception.

13.16 UnknownHappeningException

Declaration

```
public class UnknownHappeningException extends  
    java.lang.RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description

This exception is used to indicate a situation where an instance of `AsyncEvent388` attempts to bind to a happening that does not exist.

13.16.1 Constructors

```
public UnknownHappeningException()
```

A constructor for `UnknownHappeningException`.

```
public UnknownHappeningException(
    java.lang.String description)
```

A descriptive constructor for `UnknownHappeningException`.

Parameters:

`description` - Description of the error.

13.17 ResourceLimitError***Declaration***

```
public class ResourceLimitError extends java.lang.Error
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks.

13.17.1 Constructors

```
public ResourceLimitError()
```

A constructor for `ResourceLimitError`.

```
public ResourceLimitError(java.lang.String description)
```

A descriptive constructor for `ResourceLimitError`.

Parameters:

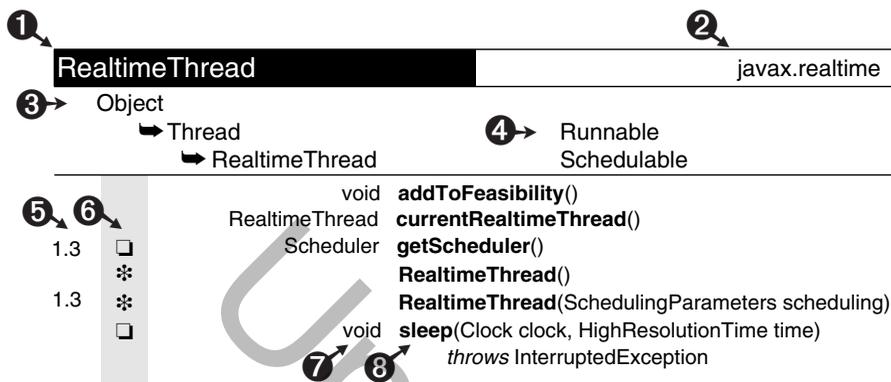
`description` - The description of the exception.

Unofficial

ALMANAC LEGEND

The almanac presents classes and interfaces in alphabetic order, regardless of their package. Fields, methods and constructors are in alphabetic order in a single list.

This almanac is modeled after the style introduced by Patrick Chan in his excellent book *Java Developers Almanac*.



1. Name of the class, interface, nested class or nested interface. Interfaces are italic.
2. Name of the package containing the class or interface.
3. Inheritance hierarchy. In this example, `RealtimeThread` extends `Thread`, which extends `Object`.
4. Implemented interfaces. The interface is to the right of, and on the same line as, the class that implements it. In this example, `Thread` implements `Runnable`, and `RealtimeThread` implements `Schedulable`.
5. The first column above is for the value of the `@since` comment, which indicates the version in which the item was introduced.
6. The second column above is for the following icons. If the “protected” symbol also has no symbols.) One symbol from each group can appear in this column.

Modifiers

- abstract
- final
- static
- static final

Access Modifiers

- ◆ protected

Constructors and Fields

- * constructor
- ⌘ field

7. Return type of a method or declared type of a field. Blank for constructors.
8. Name of the constructor, field or method. Nested classes are listed in 1, not here.

AbsoluteTime

javax.realtime

Object

↳ HighResolutionTime

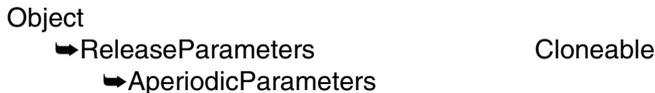
↳ AbsoluteTime

Comparable, Cloneable

		AbsoluteTime absolute (Clock clock)
		AbsoluteTime absolute (Clock clock, AbsoluteTime dest)
	*	AbsoluteTime ()
	*	AbsoluteTime (AbsoluteTime time)
1.0.1	*	AbsoluteTime (AbsoluteTime time, Clock clock)
1.0.1	*	AbsoluteTime (Clock clock)
	*	AbsoluteTime (java.util.Date date)
1.0.1	*	AbsoluteTime (java.util.Date date, Clock clock)
	*	AbsoluteTime (long millis, int nanos)
1.0.1	*	AbsoluteTime (long millis, int nanos, Clock clock)
		AbsoluteTime add (long millis, int nanos)
		AbsoluteTime add (long millis, int nanos, AbsoluteTime dest)
		AbsoluteTime add (RelativeTime time)
		AbsoluteTime add (RelativeTime time, AbsoluteTime dest)
		java.util.Date getDate ()
		RelativeTime relative (Clock clock)
		RelativeTime relative (Clock clock, RelativeTime dest)

	void set (java.util.Date date)
	RelativeTime subtract (AbsoluteTime time)
	RelativeTime subtract (AbsoluteTime time, RelativeTime dest)
	AbsoluteTime subtract (RelativeTime time)
	AbsoluteTime subtract (RelativeTime time, AbsoluteTime dest)
	String toString ()

AperiodicParameters	javax.realtime
----------------------------	-----------------------



1.0.1	*	AperiodicParameters ()
	*	AperiodicParameters (RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)
1.0.1	🗑️	String arrivalTimeQueueOverflowExcept
1.0.1	🗑️	String arrivalTimeQueueOverflowIgnore
1.0.1	🗑️	String arrivalTimeQueueOverflowReplace
1.0.1	🗑️	String arrivalTimeQueueOverflowSave
1.0.1		String getArrivalTimeQueueOverflowBehavior ()
1.0.1		int getInitialArrivalTimeQueueLength ()
1.0.1		void setArrivalTimeQueueOverflowBehavior (String behavior)
		void setDeadline (RelativeTime deadline)
		boolean setIfFeasible (RelativeTime cost, RelativeTime deadline)
1.0.1		void setInitialArrivalTimeQueueLength (int initial)

ArrivalTimeQueueOverflowException	javax.realtime
--	-----------------------



	*	ArrivalTimeQueueOverflowException ()
	*	ArrivalTimeQueueOverflowException (String description)

AsyncEvent

javax.realtime

Object

↳ AsyncEvent

*

void **addHandler**(AsyncEventHandler handler)**AsyncEvent**()void **bindTo**(String happening)ReleaseParameters **createReleaseParameters**()void **fire**()boolean **handledBy**(AsyncEventHandler handler)void **removeHandler**(AsyncEventHandler handler)void **setHandler**(AsyncEventHandler handler)void **unbindTo**(String happening)**AsyncEventHandler**

javax.realtime

Object

↳ AsyncEventHandler

Schedulable

boolean **addIfFeasible**()boolean **addToFeasibility**()

*

AsyncEventHandler()

*

AsyncEventHandler(boolean nonheap)

*

AsyncEventHandler(boolean nonheap, Runnable logic)

*

AsyncEventHandler(Runnable logic)

*

AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap)

*

AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap, Runnable logic)

*

AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic)

◆

int **getAndClearPendingFireCount**()

◆

int **getAndDecrementPendingFireCount**()

◆

int **getAndIncrementPendingFireCount**()MemoryArea **getMemoryArea**()

		MemoryParameters	getMemoryParameters()
	◆		int getPendingFireCount()
		ProcessingGroupParameters	getProcessingGroupParameters()
		ReleaseParameters	getReleaseParameters()
		Scheduler	getScheduler()
		SchedulingParameters	getSchedulingParameters()
			void handleAsyncEvent()
1.0.1	●		boolean isDaemon()
			boolean removeFromFeasibility()
	●		void run()
1.0.1	●		void setDaemon (boolean on)
			boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory)
			boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
			boolean setIfFeasible (ReleaseParameters release, ProcessingGroupParameters group)
			boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory)
			boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
			void setMemoryParameters (MemoryParameters memory)
			boolean setMemoryParametersIfFeasible (MemoryParameters memory)
			void setProcessingGroupParameters (ProcessingGroupParameters group)
			boolean setProcessingGroupParametersIfFeasible (ProcessingGroupParameters group)
			void setReleaseParameters (ReleaseParameters release)
			boolean setReleaseParametersIfFeasible (ReleaseParameters release)
			void setScheduler (Scheduler scheduler)
			void setScheduler (Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
			void setSchedulingParameters (SchedulingParameters scheduling)
			boolean setSchedulingParametersIfFeasible (SchedulingParameters scheduling)

**AsynchronouslyInterruptedEx-
ception****javax.realtime**

Object

↳ Throwable

java.io.Serializable

↳ Exception

↳ InterruptedException

↳ AsynchronouslyInterruptedException

1.0.1	*	AsynchronouslyInterruptedException()
		boolean clear()
		boolean disable()
		boolean doInterruptible (InterruptedException logic)
		boolean enable()
		boolean fire()
	□	AsynchronouslyInterruptedException getGeneric()
		boolean happened (boolean propagate)
		boolean isEnabled()
	□	void propagate()

BoundAsyncEventHandler**javax.realtime**

Object

↳ AsyncEventHandler

Schedulable

↳ BoundAsyncEventHandler

	*	BoundAsyncEventHandler()
	*	BoundAsyncEventHandler (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap, Runnable logic)

CeilingViolationException**javax.realtime**

Object

↳ Throwable

java.io.Serializable

↳ Exception

↳ RuntimeException

↳ IllegalArgumentException

↳ IllegalThreadStateException

↳ CeilingViolationException

		int getCallerPriority()
		int getCeiling()

Clock	javax.realtime
--------------	-----------------------

Object

↳Clock

	*		Clock()
1.0.1	○	RelativeTime	getEpochOffset()
	□		Clock getRealtimeClock()
	○	RelativeTime	getResolution()
	○	AbsoluteTime	getTime()
1.0.1	○	AbsoluteTime	getTime(AbsoluteTime dest)
	○		void setResolution (RelativeTime resolution)

DuplicateFilterException	javax.realtime
---------------------------------	-----------------------

Object

↳Throwable

java.io.Serializable

↳Exception

↳DuplicateFilterException

	*		DuplicateFilterException()
	*		DuplicateFilterException (String description)

GarbageCollector	javax.realtime
-------------------------	-----------------------

Object

↳GarbageCollector

	*		GarbageCollector()
	○	RelativeTime	getPreemptionLatency()

HeapMemory	javax.realtime
-------------------	-----------------------

Object

↳MemoryArea

↳HeapMemory

			void executeInArea (Runnable logic)
	□	HeapMemory	instance()

HighResolutionTime	javax.realtime
---------------------------	-----------------------

	Object		
	↳	HighResolutionTime	Comparable, Cloneable
	<input type="radio"/>	AbsoluteTime	absolute (Clock clock)
	<input type="radio"/>	AbsoluteTime	absolute (Clock clock, AbsoluteTime dest)
1.0.1		Object	clone ()
			int compareTo (HighResolutionTime time)
			int compareTo (Object object)
			boolean equals (HighResolutionTime time)
			boolean equals (Object object)
1.0.1		Clock	getClock ()
	<input checked="" type="radio"/>		long getMilliseconds ()
	<input checked="" type="radio"/>		int getNanoseconds ()
			int hashCode ()
	<input type="radio"/>	RelativeTime	relative (Clock clock)
	<input type="radio"/>	RelativeTime	relative (Clock clock, RelativeTime dest)
1.0.1			void set (HighResolutionTime time)
			void set (long millis)
			void set (long millis, int nanos)
	<input type="checkbox"/>		void waitForObject (Object target, HighResolutionTime time) <i>throws InterruptedException</i>

IllegalAssignmentError	javax.realtime
-------------------------------	-----------------------

	Object		
	↳	Throwable	java.io.Serializable
	↳	Error	
	↳	IllegalAssignmentError	
	<input checked="" type="checkbox"/>		IllegalAssignmentError ()
	<input checked="" type="checkbox"/>		IllegalAssignmentError (String description)

ImmortalMemory	javax.realtime
-----------------------	-----------------------

	Object		
	↳	MemoryArea	
	↳	ImmortalMemory	
	<input type="checkbox"/>		void executeInArea (Runnable logic)
	<input type="checkbox"/>	ImmortalMemory	instance ()

ImmutablePhysicalMemory	javax.realtime
--------------------------------	-----------------------

Object

- ↳ MemoryArea
- ↳ ImmutablePhysicalMemory

*	ImmutablePhysicalMemory (Object type, long size)
*	ImmutablePhysicalMemory (Object type, long base, long size)
*	ImmutablePhysicalMemory (Object type, long base, long size, Runnable logic)
*	ImmutablePhysicalMemory (Object type, long size, Runnable logic)
*	ImmutablePhysicalMemory (Object type, long base, SizeEstimator size)
*	ImmutablePhysicalMemory (Object type, long base, SizeEstimator size, Runnable logic)
*	ImmutablePhysicalMemory (Object type, SizeEstimator size)
*	ImmutablePhysicalMemory (Object type, SizeEstimator size, Runnable logic)

ImportanceParameters	javax.realtime
-----------------------------	-----------------------

Object

- ↳ SchedulingParameters
- ↳ PriorityParameters
- ↳ ImportanceParameters

Cloneable

	int getImportance ()
*	ImportanceParameters (int priority, int importance)
	void setImportance (int importance)
	String toString ()

InaccessibleAreaException	javax.realtime
----------------------------------	-----------------------

Object

- ↳ Throwable
- ↳ Exception
- ↳ RuntimeException
- ↳ InaccessibleAreaException

java.io.Serializable

*	InaccessibleAreaException ()
*	InaccessibleAreaException (String description)

InterruptedException		javax.realtime
InterruptedException		
		void interruptAction (AsynchronouslyInterruptedException exception)
		void run (AsynchronouslyInterruptedException exception) <i>throws</i> AsynchronouslyInterruptedException

LTMemory		javax.realtime
Object		
	↳MemoryArea	
	↳ScopedMemory	
	↳LTMemory	
1.0.1	*	LTMemory (long size)
	*	LTMemory (long initial, long maximum)
	*	LTMemory (long initial, long maximum, Runnable logic)
1.0.1	*	LTMemory (long size, Runnable logic)
1.0.1	*	LTMemory (SizeEstimator size)
1.0.1	*	LTMemory (SizeEstimator size, Runnable logic)
	*	LTMemory (SizeEstimator initial, SizeEstimator maximum)
	*	LTMemory (SizeEstimator initial, SizeEstimator maximum, Runnable logic)
		String toString ()

LTPhysicalMemory		javax.realtime
Object		
	↳MemoryArea	
	↳ScopedMemory	
	↳LTPhysicalMemory	
	*	LTPhysicalMemory (Object type, long size)
	*	LTPhysicalMemory (Object type, long base, long size)
	*	LTPhysicalMemory (Object type, long base, long size, Runnable logic)
	*	LTPhysicalMemory (Object type, long size, Runnable logic)
	*	LTPhysicalMemory (Object type, long base, SizeEstimator size)
	*	LTPhysicalMemory (Object type, long base, SizeEstimator size, Runnable logic)

*	LTPhysicalMemory (Object type, SizeEstimator size)
*	LTPhysicalMemory (Object type, SizeEstimator size, Runnable logic)
	String toString ()

MemoryAccessError	javax.realtime
--------------------------	-----------------------

Object
↳ Throwable java.io.Serializable
↳ Error
↳ MemoryAccessError

*	MemoryAccessError ()
*	MemoryAccessError (String description)

MemoryArea	javax.realtime
-------------------	-----------------------

Object
↳ MemoryArea

	void enter ()
	void enter (Runnable logic)
	void executeInArea (Runnable logic)
□	MemoryArea getMemoryArea (Object object)
*◆	MemoryArea (long size)
*◆	MemoryArea (long size, Runnable logic)
*◆	MemoryArea (SizeEstimator size)
*◆	MemoryArea (SizeEstimator size, Runnable logic)
	long memoryConsumed ()
	long memoryRemaining ()
	Object newArray (Class type, int number)
	Object newInstance (Class type) <i>throws</i> InstantiationException, IllegalAccessException
	Object newInstance (reflect.Constructor c, Object [] args) <i>throws</i> IllegalAccessException, InstantiationException, reflect.InvocationTargetException
	long size ()

MemoryInUseException	javax.realtime
-----------------------------	-----------------------

Object
↳ Throwable java.io.Serializable

↳Exception
 ↳RuntimeException
 ↳MemoryInUseException

*	MemoryInUseException()
*	MemoryInUseException(String description)

MemoryParameters javax.realtime

Object

↳MemoryParameters

Cloneable

1.0.1	Object clone()
	long getAllocationRate()
	long getMaxImmortal()
	long getMaxMemoryArea()
*	MemoryParameters (long maxMemoryArea, long maxImmortal)
*	MemoryParameters (long maxMemoryArea, long maxImmortal, long allocationRate)
🗑️	long NO_MAX
	void setAllocationRate (long allocationRate)
	boolean setAllocationRateIfFeasible (long allocationRate)
	boolean setMaxImmortalIfFeasible (long maximum)
	boolean setMaxMemoryAreaIfFeasible (long maximum)

MemoryScopeException javax.realtime

Object

↳Throwable

java.io.Serializable

↳Exception

↳RuntimeException

↳MemoryScopeException

*	MemoryScopeException()
*	MemoryScopeException(String description)

MemoryTypeConflictException javax.realtime

Object

↳Throwable

java.io.Serializable

↳Exception

↳RuntimeException

↳MemoryTypeConflictException

*	MemoryTypeConflictException()
*	MemoryTypeConflictException(String description)

MITViolationException	javax.realtime
------------------------------	-----------------------

Object

```

↳ Throwable                               java.io.Serializable
  ↳ Exception
    ↳ RuntimeException
      ↳ MITViolationException
  
```

*

MITViolationException()

*

MITViolationException(String description)

MonitorControl	javax.realtime
-----------------------	-----------------------

Object

↳ MonitorControl

☐

MonitorControl **getMonitorControl()**

☐

MonitorControl **getMonitorControl(Object obj)**

*◆

MonitorControl()

1.0.1

☐

MonitorControl **setMonitorControl(MonitorControl policy)**

1.0.1

☐

MonitorControl **setMonitorControl(Object obj, MonitorControl policy)**

NoHeapRealtimeThread	javax.realtime
-----------------------------	-----------------------

Object

```

↳ Thread                               Runnable
  ↳ RealtimeThread                       Schedulable
    ↳ NoHeapRealtimeThread
  
```

*

NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)

*

NoHeapRealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryArea area)

*

NoHeapRealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic)void **start()**

OffsetOutOfBoundsException	javax.realtime
-----------------------------------	-----------------------

Object

```

↳ Throwable                               java.io.Serializable
  
```

↳Exception
 ↳RuntimeException
 ↳OffsetOutOfBoundsException

*	OffsetOutOfBoundsException()
*	OffsetOutOfBoundsException (String description)

OneShotTimer	javax.realtime
---------------------	-----------------------

Object
 ↳AsyncEvent
 ↳Timer
 ↳OneShotTimer

*	OneShotTimer (HighResolutionTime time, AsyncEventHandler handler)
*	OneShotTimer (HighResolutionTime time, Clock clock, AsyncEventHandler handler)

PeriodicParameters	javax.realtime
---------------------------	-----------------------

Object
 ↳ReleaseParameters
 ↳PeriodicParameters

Cloneable

		RelativeTime getPeriod()
		HighResolutionTime getStart()
1.0.1	*	PeriodicParameters (HighResolutionTime start, RelativeTime period)
	*	PeriodicParameters (HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)
1.0.1	*	PeriodicParameters (RelativeTime period) void setDeadline (RelativeTime deadline) boolean setIfFeasible (RelativeTime period, RelativeTime cost, RelativeTime deadline) void setPeriod (RelativeTime period) void setStart (HighResolutionTime start)

PeriodicTimer javax.realtime

Object
 ↳ AsyncEvent
 ↳ Timer
 ↳ PeriodicTimer

	ReleaseParameters	createReleaseParameters()
1.0.1	Clock	getClock()
	AbsoluteTime	getFireTime()
1.0.1	AbsoluteTime	getFireTime (AbsoluteTime dest)
	RelativeTime	getInterval()
*		PeriodicTimer (HighResolutionTime start, RelativeTime interval, AsyncEventHandler handler)
*		PeriodicTimer (HighResolutionTime start, RelativeTime interval, Clock clock, AsyncEventHandler handler)
	void	setInterval (RelativeTime interval)

PhysicalMemoryManager javax.realtime

Object
 ↳ PhysicalMemoryManager

	Object	ALIGNED
	Object	BYTESWAP
	Object	DMA
1.0.1	Object	IO_PAGE
	boolean	isRemovable (long base, long size)
	boolean	isRemoved (long base, long size)
1.0.1	void	onInsertion (long base, long size, AsyncEvent ae)
	void	onInsertion (long base, long size, AsyncEventHandler aeh)
1.0.1	void	onRemoval (long base, long size, AsyncEvent ae)
	void	onRemoval (long base, long size, AsyncEventHandler aeh)
	void	registerFilter (Object name, PhysicalMemoryTypeFilter filter) <i>throws</i> DuplicateFilterException
	void	removeFilter (Object name)
	Object	SHARED
1.0.1	boolean	unregisterInsertionEvent (long base, long size, AsyncEvent ae)
1.0.1	boolean	unregisterRemovalEvent (long base, long size, AsyncEvent ae)

PhysicalMemoryTypeFilter**javax.realtime**

PhysicalMemoryTypeFilter

	boolean contains (long base, long size)
	long find (long base, long size)
	int getVMAttributes ()
	int getVMFlags ()
	void initialize (long base, long vBase, long size)
	boolean isPresent (long base, long size)
	boolean isRemovable ()
1.0.1	void onInsertion (long base, long size, AsyncEvent ae)
	void onInsertion (long base, long size, AsyncEventHandler aeh)
1.0.1	void onRemoval (long base, long size, AsyncEvent ae)
	void onRemoval (long base, long size, AsyncEventHandler aeh)
1.0.1	boolean unregisterInsertionEvent (long base, long size, AsyncEvent ae)
1.0.1	boolean unregisterRemovalEvent (long base, long size, AsyncEvent ae)
	long vFind (long base, long size)

POSIXSignalHandler**javax.realtime**

Object

↳ POSIXSignalHandler

<input type="checkbox"/>	void addHandler (int signal, AsyncEventHandler handler)
<input type="checkbox"/>	void removeHandler (int signal, AsyncEventHandler handler)
<input type="checkbox"/>	void setHandler (int signal, AsyncEventHandler handler)
<input checked="" type="checkbox"/>	int SIGABRT
<input checked="" type="checkbox"/>	int SIGALRM
<input checked="" type="checkbox"/>	int SIGBUS
<input checked="" type="checkbox"/>	int SIGCANCEL
<input checked="" type="checkbox"/>	int SIGCHLD
<input checked="" type="checkbox"/>	int SIGCLD
<input checked="" type="checkbox"/>	int SIGCONT
<input checked="" type="checkbox"/>	int SIGEMT
<input checked="" type="checkbox"/>	int SIGFPE
<input checked="" type="checkbox"/>	int SIGFREEZE
<input checked="" type="checkbox"/>	int SIGHUP
<input checked="" type="checkbox"/>	int SIGILL

	int SIGINT
	int SIGIO
	int SIGIOT
	int SIGKILL
	int SIGLOST
	int SIGLWP
	int SIGPIPE
	int SIGPOLL
	int SIGPROF
	int SIGPWR
	int SIGQUIT
	int SIGSEGV
	int SIGSTOP
	int SIGSYS
	int SIGTERM
	int SIGTHAW
	int SIGTRAP
	int SIGTSTP
	int SIGTTIN
	int SIGTTOU
	int SIGURG
	int SIGUSR1
	int SIGUSR2
	int SIGVTALRM
	int SIGWAITING
	int SIGWINCH
	int SIGXCPU
	int SIGXFSZ

PriorityCeilingEmulation

javax.realtime

Object

↳ MonitorControl

↳ PriorityCeilingEmulation

1.0.1		int getCeiling()
		int getDefaultCeiling()
1.0.1		PriorityCeilingEmulation getMaxCeiling()
1.0.1		PriorityCeilingEmulation instance (int ceiling)

PriorityInheritance		javax.realtime
Object		
↳ MonitorControl		
↳ PriorityInheritance		
<input type="checkbox"/>	PriorityInheritance	instance()

PriorityParameters		javax.realtime
Object		
↳ SchedulingParameters		
↳ PriorityParameters		
Cloneable		
<input type="checkbox"/>	int	getPriority()
*	PriorityParameters	PriorityParameters(int priority)
	void	setPriority(int priority)
	String	toString()

PriorityScheduler		javax.realtime
Object		
↳ Scheduler		
↳ PriorityScheduler		
◆	boolean	addToFeasibility(Schedulable schedulable)
	void	fireSchedulable(Schedulable schedulable)
	int	getMaxPriority()
<input type="checkbox"/>	int	getMaxPriority(Thread thread)
	int	getMinPriority()
<input type="checkbox"/>	int	getMinPriority(Thread thread)
	int	getNormPriority()
<input type="checkbox"/>	int	getNormPriority(Thread thread)
	String	getPolicyName()
<input type="checkbox"/>	PriorityScheduler	instance()
	boolean	isFeasible()
<input type="checkbox"/>	int	MAX_PRIORITY
<input type="checkbox"/>	int	MIN_PRIORITY
※◆	PriorityScheduler	PriorityScheduler()
◆	boolean	removeFromFeasibility(Schedulable schedulable)

```

boolean setIfFeasible(Schedulable schedulable,
                      ReleaseParameters release, MemoryParameters
                      memory)
boolean setIfFeasible(Schedulable schedulable,
                      ReleaseParameters release, MemoryParameters
                      memory, ProcessingGroupParameters group)
boolean setIfFeasible(Schedulable schedulable,
                      SchedulingParameters scheduling,
                      ReleaseParameters release, MemoryParameters
                      memory, ProcessingGroupParameters group)

```

ProcessingGroupParameters**javax.realtime**

Object

↳ ProcessingGroupParameters

Cloneable

1.0.1

```

Object clone()
RelativeTime getCost()
AsyncEventHandler getCostOverrunHandler()
RelativeTime getDeadline()
AsyncEventHandler getDeadlineMissHandler()
RelativeTime getPeriod()
HighResolutionTime getStart()

ProcessingGroupParameters(HighResolutionTime start,
                          RelativeTime period, RelativeTime cost,
                          RelativeTime deadline, AsyncEventHandler
                          overrunHandler, AsyncEventHandler missHandler)

void setCost(RelativeTime cost)
void setCostOverrunHandler(AsyncEventHandler handler)
void setDeadline(RelativeTime deadline)
void setDeadlineMissHandler(AsyncEventHandler handler)
boolean setIfFeasible(RelativeTime period, RelativeTime cost,
                      RelativeTime deadline)
void setPeriod(RelativeTime period)
void setStart(HighResolutionTime start)

```

✱

RationalTime**javax.realtime**

Object

↳ HighResolutionTime

Comparable, Cloneable

↳ RelativeTime

↳ RationalTime

```

AbsoluteTime absolute(Clock clock, AbsoluteTime destination)
void addInterarrivalTo(AbsoluteTime destination)
int getFrequency()

```

	RelativeTime getInterarrivalTime ()
	RelativeTime getInterarrivalTime (RelativeTime dest)
*	RationalTime (int frequency)
*	RationalTime (int frequency, long millis, int nanos)
*	RationalTime (int frequency, RelativeTime interval)
	void set (long millis, int nanos)
	void setFrequency (int frequency)
	String toString ()

RawMemoryAccess

javax.realtime

Object

↳ RawMemoryAccess

	byte getByte (long offset)
	void getBytes (long offset, byte[] [] bytes, int low, int number)
	int getInt (long offset)
	void getInts (long offset, int[] [] ints, int low, int number)
	long getLong (long offset)
	void getLongs (long offset, long[] [] longs, int low, int number)
	long getMappedAddress ()
	short getShort (long offset)
	void getShorts (long offset, short[] [] shorts, int low, int number)
	long map ()
	long map (long base)
	long map (long base, long size)
*	RawMemoryAccess (Object type, long size)
*	RawMemoryAccess (Object type, long base, long size)
	void setByte (long offset, byte value)
	void setBytes (long offset, byte[] [] bytes, int low, int number)
	void setInt (long offset, int value)
	void setInts (long offset, int[] [] ints, int low, int number)
	void setLong (long offset, long value)
	void setLongs (long offset, long[] [] longs, int low, int number)
	void setShort (long offset, short value)
	void setShorts (long offset, short[] [] shorts, int low, int number)
	void unmap ()

RawMemoryFloatAccess **javax.realtime**

Object

- ↳RawMemoryAccess
- ↳RawMemoryFloatAccess

	double getDouble (long offset)
	void getDoubles (long offset, double[] [] doubles, int low, int number)
	float getFloat (long offset)
	void getFloats (long offset, float[] [] floats, int low, int number)
*	RawMemoryFloatAccess (Object type, long size)
*	RawMemoryFloatAccess (Object type, long base, long size)
	void setDouble (long offset, double value)
	void setDoubles (long offset, double[] [] doubles, int low, int number)
	void setFloat (long offset, float value)
	void setFloats (long offset, float[] [] floats, int low, int number)

RealtimeSecurity **javax.realtime**

Object

- ↳RealtimeSecurity

	void checkAccessPhysical () <i>throws</i> SecurityException
	void checkAccessPhysicalRange (long base, long size) <i>throws</i> SecurityException
1.0.1	void checkAEHSetDaemon () <i>throws</i> SecurityException
	void checkSetFilter () <i>throws</i> SecurityException
1.0.1	void checkSetMonitorControl (MonitorControl policy) <i>throws</i> SecurityException
	void checkSetScheduler () <i>throws</i> SecurityException
*	RealtimeSecurity ()

RealtimeSystem **javax.realtime**

Object

- ↳RealtimeSystem

	byte BIG_ENDIAN
	byte BYTE_ORDER
	GarbageCollector currentGC ()
	int getConcurrentLocksUsed ()
1.0.1	MonitorControl getInitialMonitorControl ()

<input type="checkbox"/>	int getMaximumConcurrentLocks()
<input type="checkbox"/>	RealtimeSecurity getSecurityManager()
<input checked="" type="checkbox"/>	byte LITTLE_ENDIAN
<input type="checkbox"/>	void setMaximumConcurrentLocks (int numLocks)
<input type="checkbox"/>	void setMaximumConcurrentLocks (int number, boolean hard)
<input type="checkbox"/>	void setSecurityManager (RealtimeSecurity manager)

RealtimeThread**javax.realtime**

Object

↳ Thread

Runnable

↳ RealtimeThread

Schedulable

	boolean addIfFeasible()
	boolean addToFeasibility()
<input type="checkbox"/>	RealtimeThread currentRealtimeThread()
	void deschedulePeriodic()
<input type="checkbox"/>	MemoryArea getCurrentMemoryArea()
<input type="checkbox"/>	int getInitialMemoryAreaIndex()
1.0.1	MemoryArea getMemoryArea()
<input type="checkbox"/>	int getMemoryAreaStackDepth()
	MemoryParameters getMemoryParameters()
<input type="checkbox"/>	MemoryArea getOuterMemoryArea (int index)
	ProcessingGroupParameters getProcessingGroupParameters()
	ReleaseParameters getReleaseParameters()
	Scheduler getScheduler()
	SchedulingParameters getSchedulingParameters()
	void interrupt()
*	RealtimeThread()
*	RealtimeThread (SchedulingParameters scheduling)
*	RealtimeThread (SchedulingParameters scheduling, ReleaseParameters release)
*	RealtimeThread (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic)
	boolean removeFromFeasibility()
	void schedulePeriodic()
	boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory)

		boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
		boolean setIfFeasible (ReleaseParameters release, ProcessingGroupParameters group)
		boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory)
		boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
		void setMemoryParameters (MemoryParameters memory)
		boolean setMemoryParametersIfFeasible (MemoryParameters memory)
		void setProcessingGroupParameters (ProcessingGroupParameters group)
		boolean setProcessingGroupParametersIfFeasible (ProcessingGroupParameters group)
		void setReleaseParameters (ReleaseParameters release)
		boolean setReleaseParametersIfFeasible (ReleaseParameters release)
		void setScheduler (Scheduler scheduler)
		void setScheduler (Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters group)
		void setSchedulingParameters (SchedulingParameters scheduling)
		boolean setSchedulingParametersIfFeasible (SchedulingParameters scheduling)
	<input type="checkbox"/>	void sleep (Clock clock, HighResolutionTime time) <i>throws</i> InterruptedException
	<input type="checkbox"/>	void sleep (HighResolutionTime time) <i>throws</i> InterruptedException
		void start ()
1.0.1	<input type="checkbox"/>	boolean waitForNextPeriod ()
1.0.1	<input type="checkbox"/>	boolean waitForNextPeriodInterruptible () <i>throws</i> InterruptedException

RelativeTime	javax.realtime
---------------------	-----------------------

	Object		Comparable, Cloneable
	↳ HighResolutionTime		
	↳ RelativeTime		
<hr/>			
		AbsoluteTime absolute (Clock clock)	
		AbsoluteTime absolute (Clock clock, AbsoluteTime dest)	
		RelativeTime add (long millis, int nanos)	
		RelativeTime add (long millis, int nanos, RelativeTime dest)	
		RelativeTime add (RelativeTime time)	
		RelativeTime add (RelativeTime time, RelativeTime dest)	
		void addInterarrivalTo (AbsoluteTime timeAndDestination)	
		RelativeTime getInterarrivalTime ()	
		RelativeTime getInterarrivalTime (RelativeTime destination)	
		RelativeTime relative (Clock clock)	
		RelativeTime relative (Clock clock, RelativeTime dest)	
	*	RelativeTime ()	
1.0.1	*	RelativeTime (Clock clock)	
	*	RelativeTime (long millis, int nanos)	
1.0.1	*	RelativeTime (long millis, int nanos, Clock clock)	
	*	RelativeTime (RelativeTime time)	
1.0.1	*	RelativeTime (RelativeTime time, Clock clock)	
		RelativeTime subtract (RelativeTime time)	
		RelativeTime subtract (RelativeTime time, RelativeTime dest)	
		String toString ()	

ReleaseParameters	javax.realtime
--------------------------	-----------------------

	Object		Cloneable
	↳ ReleaseParameters		
<hr/>			
1.0.1		Object clone ()	
		RelativeTime getCost ()	
		AsyncEventHandler getCostOverrunHandler ()	
		RelativeTime getDeadline ()	
		AsyncEventHandler getDeadlineMissHandler ()	
	*◆	ReleaseParameters ()	
	*◆	ReleaseParameters (RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)	
		void setCost (RelativeTime cost)	

	void setCostOverrunHandler (AsyncEventHandler handler) void setDeadline (RelativeTime deadline) void setDeadlineMissHandler (AsyncEventHandler handler) boolean setIfFeasible (RelativeTime cost, RelativeTime deadline)
--	---

ResourceLimitError	javax.realtime
---------------------------	-----------------------



*	ResourceLimitError ()
*	ResourceLimitError (String description)

Schedulable	javax.realtime
--------------------	-----------------------

	Schedulable	Runnable
1.0.1	boolean addIfFeasible ()	
	boolean addToFeasibility ()	
	MemoryParameters getMemoryParameters ()	
	ProcessingGroupParameters getProcessingGroupParameters ()	
	ReleaseParameters getReleaseParameters ()	
	Scheduler getScheduler ()	
	SchedulingParameters getSchedulingParameters ()	
	boolean removeFromFeasibility ()	
1.0.1	boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory)	
1.0.1	boolean setIfFeasible (ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)	
1.0.1	boolean setIfFeasible (ReleaseParameters release, ProcessingGroupParameters group)	
1.0.1	boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory)	
1.0.1	boolean setIfFeasible (SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)	
	void setMemoryParameters (MemoryParameters memory)	
	boolean setMemoryParametersIfFeasible (MemoryParameters memory)	
	void setProcessingGroupParameters (ProcessingGroupParameters group)	

```

boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)
void setReleaseParameters(ReleaseParameters release)
boolean setReleaseParametersIfFeasible(ReleaseParameters release)
void setScheduler(Scheduler scheduler)
void setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters group)
void setSchedulingParameters(SchedulingParameters scheduling)
boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)

```

Scheduler**javax.realtime**

Object

↳ Scheduler

```

○◆ boolean addToFeasibility(Schedulable schedulable)
○ void fireSchedulable(Schedulable schedulable)
□ Scheduler getDefaultScheduler()
○ String getPolicyName()
○ boolean isFeasible()
○◆ boolean removeFromFeasibility(Schedulable schedulable)
※◆ Scheduler()
□ void setDefaultScheduler(Scheduler scheduler)
○ boolean setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)
○ boolean setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
○ boolean setIfFeasible(Schedulable schedulable, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)

```

SchedulingParameters**javax.realtime**

Object

↳ SchedulingParameters

Cloneable

```

1.0.1 Object clone()
1.0.1 ※◆ SchedulingParameters()

```

ScopedCycleException	javax.realtime
-----------------------------	-----------------------

Object

↳ Throwable

java.io.Serializable

↳ Exception

↳ RuntimeException

↳ ScopedCycleException

*

ScopedCycleException()

*

ScopedCycleException(String description)

ScopedMemory	javax.realtime
---------------------	-----------------------

Object

↳ MemoryArea

↳ ScopedMemory

void **enter**()void **enter**(Runnable logic)void **executeInArea**(Runnable logic)long **getMaximumSize**()Object **getPortal**()int **getReferenceCount**()void **join**() *throws* InterruptedExceptionvoid **join**(HighResolutionTime time) *throws*
InterruptedExceptionvoid **joinAndEnter**() *throws* InterruptedExceptionvoid **joinAndEnter**(HighResolutionTime time) *throws*
InterruptedExceptionvoid **joinAndEnter**(Runnable logic) *throws*
InterruptedExceptionvoid **joinAndEnter**(Runnable logic, HighResolutionTime time)
throws InterruptedExceptionObject **newArray**(Class type, int number)Object **newInstance**(Class type) *throws* IllegalAccessException,
InstantiationExceptionObject **newInstance**(reflect.Constructor c, Object [] args) *throws*
IllegalAccessException, InstantiationException,
reflect.InvocationTargetException

*

ScopedMemory(long size)

*

ScopedMemory(long size, Runnable logic)

*

ScopedMemory(SizeEstimator size)

*	ScopedMemory (SizeEstimator size, Runnable logic) void setPortal (Object object) String toString ()
---	--

SizeEstimator	javax.realtime
----------------------	-----------------------

Object	↳SizeEstimator	
		long getEstimate ()
		void reserve (Class c, int number)
		void reserve (SizeEstimator size)
		void reserve (SizeEstimator estimator, int number)
1.0.1		void reserveArray (int length)
1.0.1		void reserveArray (int length, Class type)
*		SizeEstimator ()

SizeOutOfBoundsException	javax.realtime
---------------------------------	-----------------------

Object	↳Throwable	↳Exception	↳RuntimeException	↳SizeOutOfBoundsException	java.io.Serializable
*					SizeOutOfBoundsException ()
*					SizeOutOfBoundsException (String description)

SporadicParameters	javax.realtime
---------------------------	-----------------------

Object	↳ReleaseParameters	↳AperiodicParameters	↳SporadicParameters	Cloneable
				RelativeTime getMinimumInterarrival ()
				String getMitViolationBehavior ()
				String mitViolationExcept
				String mitViolationIgnore
				String mitViolationReplace
				String mitViolationSave
				boolean setIfFeasible (RelativeTime cost, RelativeTime deadline)
				boolean setIfFeasible (RelativeTime interarrival, RelativeTime cost, RelativeTime deadline)
				void setMinimumInterarrival (RelativeTime minimum)

1.0.1	* *	void setMitViolationBehavior (String behavior) SporadicParameters (RelativeTime minInterarrival) SporadicParameters (RelativeTime minInterarrival, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)
-------	--------	---

ThrowBoundaryError **javax.realtime**



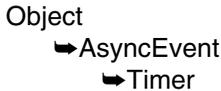
*	ThrowBoundaryError ()
*	ThrowBoundaryError (String description)

Timed **javax.realtime**



*	boolean doInterruptible (InterruptedException logic) void resetTime (HighResolutionTime time) Timed (HighResolutionTime time)
---	--

Timer **javax.realtime**



1.0.1	void addHandler (AsyncEventHandler handler)
1.0.1	void bindTo (String happening)
	ReleaseParameters createReleaseParameters ()
	void destroy ()
	void disable ()
	void enable ()
1.0.1	void fire ()
	Clock getClock ()
	AbsoluteTime getFireTime ()

1.0.1		AbsoluteTime getFireTime (AbsoluteTime dest)
1.0.1		boolean handledBy (AsyncEventHandler handler)
		boolean isRunning ()
1.0.1		void removeHandler (AsyncEventHandler handler)
		void reschedule (HighResolutionTime time)
1.0.1		void setHandler (AsyncEventHandler handler)
		void start ()
1.0.1		void start (boolean disabled)
		boolean stop ()
	❖	Timer (HighResolutionTime time, Clock clock, AsyncEventHandler handler)

UnknownHappeningException javax.realtime

Object
 ↳ Throwable java.io.Serializable
 ↳ Exception
 ↳ RuntimeException
 ↳ UnknownHappeningException

❖	UnknownHappeningException ()
❖	UnknownHappeningException (String description)

UnsupportedPhysicalMemoryException javax.realtime

Object
 ↳ Throwable java.io.Serializable
 ↳ Exception
 ↳ RuntimeException
 ↳ UnsupportedPhysicalMemoryException

❖	UnsupportedPhysicalMemoryException ()
❖	UnsupportedPhysicalMemoryException (String description)

VTMemory javax.realtime

Object
 ↳ MemoryArea
 ↳ ScopedMemory
 ↳ VTMemory

		String toString ()
1.0.1	❖	VTMemory (long size)
	❖	VTMemory (long initial, long maximum)

	*	VTMemory (long initial, long maximum, Runnable logic)
1.0.1	*	VTMemory (long size, Runnable logic)
1.0.1	*	VTMemory (SizeEstimator size)
1.0.1	*	VTMemory (SizeEstimator size, Runnable logic)
	*	VTMemory (SizeEstimator initial, SizeEstimator maximum)
	*	VTMemory (SizeEstimator initial, SizeEstimator maximum, Runnable logic)

VTPhysicalMemory

javax.realtime

Object

↳MemoryArea

↳ScopedMemory

↳VTPhysicalMemory

		String toString ()
	*	VTPhysicalMemory (Object type, long size)
	*	VTPhysicalMemory (Object type, long base, long size)
	*	VTPhysicalMemory (Object type, long base, long size, Runnable logic)
	*	VTPhysicalMemory (Object type, long size, Runnable logic)
	*	VTPhysicalMemory (Object type, long base, SizeEstimator size)
	*	VTPhysicalMemory (Object type, long base, SizeEstimator size, Runnable logic)
	*	VTPhysicalMemory (Object type, SizeEstimator size)
	*	VTPhysicalMemory (Object type, SizeEstimator size, Runnable logic)

WaitFreeDeque

javax.realtime

Object

↳WaitFreeDeque

1.0.1		Object blockingRead () <i>throws</i> InterruptedException
1.0.1		void blockingWrite (Object object) <i>throws</i> InterruptedException
		boolean force (Object object)
		Object nonBlockingRead ()
		boolean nonBlockingWrite (Object object)
	*	WaitFreeDeque (Runnable writer, Runnable reader, int maximum, MemoryArea memory)

WaitFreeReadQueue	javax.realtime
--------------------------	-----------------------

Object

↳WaitFreeReadQueue

		void clear()	
		boolean isEmpty()	
		boolean isFull()	
		Object read()	
		int size()	
1.0.1		void waitForData() <i>throws</i> InterruptedException	
1.0.1	*	WaitFreeReadQueue (int maximum, boolean notify)	
1.0.1	*	WaitFreeReadQueue (int maximum, MemoryArea memory, boolean notify)	
	*	WaitFreeReadQueue (Runnable writer, Runnable reader, int maximum, MemoryArea memory)	
	*	WaitFreeReadQueue (Runnable writer, Runnable reader, int maximum, MemoryArea memory, boolean notify)	
1.0.1		void write (Object object) <i>throws</i> InterruptedException	

WaitFreeWriteQueue	javax.realtime
---------------------------	-----------------------

Object

↳WaitFreeWriteQueue

		void clear()	
		boolean force (Object object)	
		boolean isEmpty()	
		boolean isFull()	
1.0.1		Object read() <i>throws</i> InterruptedException	
		int size()	
1.0.1	*	WaitFreeWriteQueue (int maximum)	
1.0.1	*	WaitFreeWriteQueue (int maximum, MemoryArea memory)	
	*	WaitFreeWriteQueue (Runnable writer, Runnable reader, int maximum, MemoryArea memory)	
		boolean write (Object object)	

Unofficial

Index

A

absolute(Clock)

- of `javax.realtime.AbsoluteTime` 326
- of `javax.realtime.HighResolutionTime` 316
- of `javax.realtime.RelativeTime` 337

absolute(Clock, AbsoluteTime)

- of `javax.realtime.AbsoluteTime` 326
- of `javax.realtime.HighResolutionTime` 317
- of `javax.realtime.RationalTime` 345
- of `javax.realtime.RelativeTime` 337

AbsoluteTime

- of `javax.realtime` 322

AbsoluteTime()

- of `javax.realtime.AbsoluteTime` 323

AbsoluteTime(AbsoluteTime)

- of `javax.realtime.AbsoluteTime` 323

AbsoluteTime(AbsoluteTime, Clock)

- of `javax.realtime.AbsoluteTime` 323

AbsoluteTime(Clock)

- of `javax.realtime.AbsoluteTime` 324

AbsoluteTime(Date)

- of `javax.realtime.AbsoluteTime` 324

AbsoluteTime(Date, Clock)

- of `javax.realtime.AbsoluteTime` 324

AbsoluteTime(long, int)

- of `javax.realtime.AbsoluteTime` 325

AbsoluteTime(long, int, Clock)

- of `javax.realtime.AbsoluteTime` 325

add(long, int)

- of `javax.realtime.AbsoluteTime` 327
- of `javax.realtime.RelativeTime` 338

add(long, int, AbsoluteTime)

- of `javax.realtime.AbsoluteTime` 327

add(long, int, RelativeTime)

- of `javax.realtime.RelativeTime` 338

add(RelativeTime)

- of `javax.realtime.AbsoluteTime` 328
- of `javax.realtime.RelativeTime` 339

add(RelativeTime, AbsoluteTime)

- of `javax.realtime.AbsoluteTime` 329

add(RelativeTime, RelativeTime)

- of `javax.realtime.RelativeTime` 340

addHandler(AsyncEventHandler)

- of `javax.realtime.AsyncEvent` 391
- of `javax.realtime.Timer` 364

addHandler(int, AsyncEventHandler)

- of `javax.realtime.POSIXSignalHandler` 439

addIfFeasible()

- of `javax.realtime.AsyncEventHandler` 400
- of `javax.realtime.RealtimeThread` 32
- of `javax.realtime.Schedulable` 81

addInterarrivalTo(AbsoluteTime)

- of `javax.realtime.RationalTime` 346
- of `javax.realtime.RelativeTime` 340

addToFeasibility()

- of `javax.realtime.AsyncEventHandler` 400
- of `javax.realtime.RealtimeThread` 32
- of `javax.realtime.Schedulable` 81

addToFeasibility(Schedulable)

- of `javax.realtime.PriorityScheduler` 104
- of `javax.realtime.Scheduler` 98

ALIGNED

- of `javax.realtime.PhysicalMemoryManager` 200

AperiodicParameters

- of `javax.realtime` 130

AperiodicParameters()

- of `javax.realtime.AperiodicParameters` 133

AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

- of `javax.realtime.AperiodicParameters` 133

arrivalTimeQueueOverflowExcept

- of `javax.realtime.AperiodicParameters` 132

ArrivalTimeQueueOverflowException
of javax.realtime 448

ArrivalTimeQueueOverflowException()
of javax.realtime.ArrivalTimeQueueOverflowException 448

ArrivalTimeQueueOverflowException(String)
of javax.realtime.ArrivalTimeQueueOverflowException 448

arrivalTimeQueueOverflowIgnore
of javax.realtime.AperiodicParameters 132

arrivalTimeQueueOverflowReplace
of javax.realtime.AperiodicParameters 133

arrivalTimeQueueOverflowSave
of javax.realtime.AperiodicParameters 133

AsyncEvent
of javax.realtime 390

AsyncEvent()
of javax.realtime.AsyncEvent 391

AsyncEventHandler
of javax.realtime 395

AsyncEventHandler()
of javax.realtime.AsyncEventHandler 396

AsyncEventHandler(boolean)
of javax.realtime.AsyncEventHandler 396

AsyncEventHandler(boolean, Runnable)
of javax.realtime.AsyncEventHandler 396

AsyncEventHandler(Runnable)
of javax.realtime.AsyncEventHandler 397

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean)
of javax.realtime.AsyncEventHandler 397

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean, Runnable)
of javax.realtime.AsyncEventHandler

398

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)
of javax.realtime.AsyncEventHandler 399

AsynchronouslyInterruptedException
of javax.realtime 424

AsynchronouslyInterruptedException()
of javax.realtime.AsynchronouslyInterruptedException 425

B

BIG_ENDIAN
of javax.realtime.RealtimeSystem 443

bindTo(String)
of javax.realtime.AsyncEvent 391
of javax.realtime.Timer 365

blockingRead()
of javax.realtime.WaitFreeDequeue 307

blockingWrite(Object)
of javax.realtime.WaitFreeDequeue 307

BoundAsyncEventHandler
of javax.realtime 421

BoundAsyncEventHandler()
of javax.realtime.BoundAsyncEventHandler 421

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean, Runnable)
of javax.realtime.BoundAsyncEventHandler 421

BYTE_ORDER
of javax.realtime.RealtimeSystem 443

BYTESWAP
of javax.realtime.PhysicalMemoryManager 200

C

CeilingViolationException
of javax.realtime 449

checkAccessPhysical()
of javax.realtime.RealtimeSecurity 441

- checkAccessPhysicalRange(long, long)**
 - of javax.realtime.RealtimeSecurity 442
 - checkAEHSetDaemon()**
 - of javax.realtime.RealtimeSecurity 442
 - checkSetFilter()**
 - of javax.realtime.RealtimeSecurity 442
 - checkSetMonitorControl(MonitorControl)**
 - of javax.realtime.RealtimeSecurity 442
 - checkSetScheduler()**
 - of javax.realtime.RealtimeSecurity 443
 - clear()**
 - of javax.realtime.AsynchronouslyInterruptedException 425
 - of javax.realtime.WaitFreeReadQueue 303
 - of javax.realtime.WaitFreeWriteQueue 297
 - Clock**
 - of javax.realtime 354
 - Clock()**
 - of javax.realtime.Clock 354
 - clone()**
 - of javax.realtime.HighResolutionTime 317
 - of javax.realtime.MemoryParameters 277
 - of javax.realtime.ProcessingGroupParameters 147
 - of javax.realtime.ReleaseParameters 119
 - of javax.realtime.SchedulingParameters 113
 - compareTo(HighResolutionTime)**
 - of javax.realtime.HighResolutionTime 317
 - compareTo(Object)**
 - of javax.realtime.HighResolutionTime 318
 - contains(long, long)**
 - of javax.realtime.PhysicalMemoryTypeFilter 207
 - createReleaseParameters()**
 - of javax.realtime.AsyncEvent 392
 - of javax.realtime.PeriodicTimer 376
 - of javax.realtime.Timer 365
 - currentGC()**
 - of javax.realtime.RealtimeSystem 444
 - currentRealtimeThread()**
 - of javax.realtime.RealtimeThread 33
- D**
- deschedulePeriodic()**
 - of javax.realtime.RealtimeThread 33
 - destroy()**
 - of javax.realtime.Timer 366
 - disable()**
 - of javax.realtime.AsynchronouslyInterruptedException 426
 - of javax.realtime.Timer 366
 - DMA**
 - of javax.realtime.PhysicalMemoryManager 201
 - doInterruptible(Interruptible)**
 - of javax.realtime.AsynchronouslyInterruptedException 426
 - of javax.realtime.Timed 431
 - DuplicateFilterException**
 - of javax.realtime 449
 - DuplicateFilterException()**
 - of javax.realtime.DuplicateFilterException 450
 - DuplicateFilterException(String)**
 - of javax.realtime.DuplicateFilterException 450
- E**
- enable()**
 - of javax.realtime.AsynchronouslyInterruptedException 427
 - of javax.realtime.Timer 366
 - enter()**
 - of javax.realtime.MemoryArea 165
 - of javax.realtime.ScopedMemory 177
 - enter(Runnable)**
 - of javax.realtime.MemoryArea 166
 - of javax.realtime.ScopedMemory 178
 - equals(HighResolutionTime)**
 - of javax.realtime.HighResolutionTime 318
 - equals(Object)**
 - of javax.realtime.HighResolutionTime 318
 - executeInArea(Runnable)**
 - of javax.realtime.HeapMemory 170
 - of javax.realtime.ImmortalMemory 171
 - of javax.realtime.MemoryArea 166
 - of javax.realtime.ScopedMemory 178

F**find(long, long)**

of `javax.realtime.PhysicalMemoryTypeFilter` 208

fire()

of `javax.realtime.AsyncEvent` 392
 of `javax.realtime.AsynchronouslyInterruptedException` 427
 of `javax.realtime.Timer` 367

fireSchedulable(Schedulable)

of `javax.realtime.PriorityScheduler` 105
 of `javax.realtime.Scheduler` 98

force(Object)

of `javax.realtime.WaitFreeDeque` 308
 of `javax.realtime.WaitFreeWriteQueue` 297

G**GarbageCollector**

of `javax.realtime` 280

GarbageCollector()

of `javax.realtime.GarbageCollector` 280

getAllocationRate()

of `javax.realtime.MemoryParameters` 277

getAndClearPendingFireCount()

of `javax.realtime.AsyncEventHandler` 400

getAndDecrementPendingFireCount()

of `javax.realtime.AsyncEventHandler` 401

getAndIncrementPendingFireCount()

of `javax.realtime.AsyncEventHandler` 401

getArrivalTimeQueueOverflowBehavior()

of `javax.realtime.AperiodicParameters` 134

getByte(long)

of `javax.realtime.RawMemoryAccess` 246

getBytes(long, byte[], int, int)

of `javax.realtime.RawMemoryAccess` 247

getCallerPriority()

of `javax.realtime.CeilingViolationException` 449

getCeiling()

of `javax.realtime.CeilingViolationException`

tion 449

of `javax.realtime.PriorityCeilingEmulation` 292

getClock()

of `javax.realtime.HighResolutionTime` 318

of `javax.realtime.PeriodicTimer` 376

of `javax.realtime.Timer` 367

getConcurrentLocksUsed()

of `javax.realtime.RealtimeSystem` 444

getCost()

of `javax.realtime.ProcessingGroupParameters` 147

of `javax.realtime.ReleaseParameters` 120

getCostOverrunHandler()

of `javax.realtime.ProcessingGroupParameters` 148

of `javax.realtime.ReleaseParameters` 120

getCurrentMemoryArea()

of `javax.realtime.RealtimeThread` 33

getDate()

of `javax.realtime.AbsoluteTime` 329

getDeadline()

of `javax.realtime.ProcessingGroupParameters` 148

of `javax.realtime.ReleaseParameters` 120

getDeadlineMissHandler()

of `javax.realtime.ProcessingGroupParameters` 148

of `javax.realtime.ReleaseParameters` 120

getDefaultCeiling()

of `javax.realtime.PriorityCeilingEmulation` 292

getDefaultScheduler()

of `javax.realtime.Scheduler` 98

getDouble(long)

of `javax.realtime.RawMemoryFloatAccess` 267

getDoubles(long, double[], int, int)

of `javax.realtime.RawMemoryFloatAccess` 268

getEpochOffset()

of `javax.realtime.Clock` 354

getEstimate()

of `javax.realtime.SizeEstimator` 172

getFireTime()

of `javax.realtime.PeriodicTimer` 377

of `javax.realtime.Timer` 367

- getFireTime(AbsoluteTime)**
 - of javax.realtime.PeriodicTimer 377
 - of javax.realtime.Timer 368
- getFloat(long)**
 - of javax.realtime.RawMemoryFloatAccess 269
- getFloats(long, float[], int, int)**
 - of javax.realtime.RawMemoryFloatAccess 270
- getFrequency()**
 - of javax.realtime.RationalTime 346
- getGeneric()**
 - of javax.realtime.AsynchronouslyInterruptedException 427
- getImportance()**
 - of javax.realtime.ImportanceParameters 116
- getInitialArrivalTimeQueueLength()**
 - of javax.realtime.AperiodicParameters 135
- getInitialMemoryAreaIndex()**
 - of javax.realtime.RealtimeThread 33
- getInitialMonitorControl()**
 - of javax.realtime.RealtimeSystem 444
- getInt(long)**
 - of javax.realtime.RawMemoryAccess 248
- getInterarrivalTime()**
 - of javax.realtime.RationalTime 346
 - of javax.realtime.RelativeTime 341
- getInterarrivalTime(RelativeTime)**
 - of javax.realtime.RationalTime 346
 - of javax.realtime.RelativeTime 341
- getInterval()**
 - of javax.realtime.PeriodicTimer 378
- getInts(long, int[], int, int)**
 - of javax.realtime.RawMemoryAccess 249
- getLong(long)**
 - of javax.realtime.RawMemoryAccess 250
- getLongs(long, long[], int, int)**
 - of javax.realtime.RawMemoryAccess 250
- getMappedAddress()**
 - of javax.realtime.RawMemoryAccess 252
- getMaxCeiling()**
 - of javax.realtime.PriorityCeilingEmulation 292
- getMaxImmortal()**
 - of javax.realtime.MemoryParameters 277
- getMaximumConcurrentLocks()**
 - of javax.realtime.RealtimeSystem 444
- getMaximumSize()**
 - of javax.realtime.ScopedMemory 179
- getMaxMemoryArea()**
 - of javax.realtime.MemoryParameters 278
- getMaxPriority()**
 - of javax.realtime.PriorityScheduler 105
- getMaxPriority(Thread)**
 - of javax.realtime.PriorityScheduler 105
- getMemoryArea()**
 - of javax.realtime.AsyncEventHandler 402
 - of javax.realtime.RealtimeThread 34
- getMemoryArea(Object)**
 - of javax.realtime.MemoryArea 167
- getMemoryAreaStackDepth()**
 - of javax.realtime.RealtimeThread 34
- getMemoryParameters()**
 - of javax.realtime.AsyncEventHandler 402
 - of javax.realtime.RealtimeThread 34
 - of javax.realtime.Schedulable 82
- getMilliseconds()**
 - of javax.realtime.HighResolutionTime 319
- getMinimumInterarrival()**
 - of javax.realtime.SporadicParameters 141
- getMinPriority()**
 - of javax.realtime.PriorityScheduler 106
- getMinPriority(Thread)**
 - of javax.realtime.PriorityScheduler 106
- getMitViolationBehavior()**
 - of javax.realtime.SporadicParameters 141
- getMonitorControl()**
 - of javax.realtime.MonitorControl 288
- getMonitorControl(Object)**
 - of javax.realtime.MonitorControl 288
- getNanoseconds()**
 - of javax.realtime.HighResolutionTime 319
- getNormPriority()**
 - of javax.realtime.PriorityScheduler 106

- getNormPriority(Thread)**
 - of javax.realtime.PriorityScheduler 107
 - getOuterMemoryArea(int)**
 - of javax.realtime.RealtimeThread 35
 - getPendingFireCount()**
 - of javax.realtime.AsyncEventHandler 402
 - getPeriod()**
 - of javax.realtime.PeriodicParameters 127
 - of javax.realtime.ProcessingGroupParameters 148
 - getPolicyName()**
 - of javax.realtime.PriorityScheduler 107
 - of javax.realtime.Scheduler 98
 - getPortal()**
 - of javax.realtime.ScopedMemory 179
 - getPreemptionLatency()**
 - of javax.realtime.GarbageCollector 280
 - getPriority()**
 - of javax.realtime.PriorityParameters 114
 - getProcessingGroupParameters()**
 - of javax.realtime.AsyncEventHandler 403
 - of javax.realtime.RealtimeThread 35
 - of javax.realtime.Schedulable 82
 - getRealtimeClock()**
 - of javax.realtime.Clock 355
 - getReferenceCount()**
 - of javax.realtime.ScopedMemory 179
 - getReleaseParameters()**
 - of javax.realtime.AsyncEventHandler 403
 - of javax.realtime.RealtimeThread 35
 - of javax.realtime.Schedulable 82
 - getResolution()**
 - of javax.realtime.Clock 355
 - getScheduler()**
 - of javax.realtime.AsyncEventHandler 403
 - of javax.realtime.RealtimeThread 36
 - of javax.realtime.Schedulable 82
 - getSchedulingParameters()**
 - of javax.realtime.AsyncEventHandler 403
 - of javax.realtime.RealtimeThread 36
 - of javax.realtime.Schedulable 82
 - getSecurityManager()**
 - of javax.realtime.RealtimeSystem 445
 - getShort(long)**
 - of javax.realtime.RawMemoryAccess 252
 - getShorts(long, short[], int, int)**
 - of javax.realtime.RawMemoryAccess 253
 - getStart()**
 - of javax.realtime.PeriodicParameters 128
 - of javax.realtime.ProcessingGroupParameters 148
 - getTime()**
 - of javax.realtime.Clock 355
 - getTime(AbsoluteTime)**
 - of javax.realtime.Clock 355
 - getVMAttributes()**
 - of javax.realtime.PhysicalMemoryTypeFilter 208
 - getVMFlags()**
 - of javax.realtime.PhysicalMemoryTypeFilter 209
- ## H
- handleAsyncEvent()**
 - of javax.realtime.AsyncEventHandler 403
 - handledBy(AsyncEventHandler)**
 - of javax.realtime.AsyncEvent 393
 - of javax.realtime.Timer 368
 - happened(boolean)**
 - of javax.realtime.AsynchronouslyInterruptedException 428
 - hashCode()**
 - of javax.realtime.HighResolutionTime 319
 - HeapMemory**
 - of javax.realtime 170
 - HighResolutionTime**
 - of javax.realtime 316
- ## I
- IllegalAssignmentError**
 - of javax.realtime 450
 - IllegalAssignmentError()**
 - of javax.realtime.IllegalAssignmentError 450
 - IllegalAssignmentError(String)**
 - of javax.realtime.IllegalAssignmentError

- 450
- ImmortalMemory**
 - of javax.realtime 170
- ImmortalPhysicalMemory**
 - of javax.realtime 214
- ImmortalPhysicalMemory(Object, long)**
 - of javax.realtime.ImmortalPhysicalMemory 215
- ImmortalPhysicalMemory(Object, long, long)**
 - of javax.realtime.ImmortalPhysicalMemory 216
- ImmortalPhysicalMemory(Object, long, long, Runnable)**
 - of javax.realtime.ImmortalPhysicalMemory 217
- ImmortalPhysicalMemory(Object, long, Runnable)**
 - of javax.realtime.ImmortalPhysicalMemory 218
- ImmortalPhysicalMemory(Object, long, SizeEstimator)**
 - of javax.realtime.ImmortalPhysicalMemory 219
- ImmortalPhysicalMemory(Object, long, SizeEstimator, Runnable)**
 - of javax.realtime.ImmortalPhysicalMemory 220
- ImmortalPhysicalMemory(Object, SizeEstimator)**
 - of javax.realtime.ImmortalPhysicalMemory 221
- ImmortalPhysicalMemory(Object, SizeEstimator, Runnable)**
 - of javax.realtime.ImmortalPhysicalMemory 222
- ImportanceParameters**
 - of javax.realtime 115
- ImportanceParameters(int, int)**
 - of javax.realtime.ImportanceParameters 115
- InaccessibleAreaException**
 - of javax.realtime 451
- InaccessibleAreaException()**
 - of javax.realtime.InaccessibleAreaException 451
- InaccessibleAreaException(String)**
 - of javax.realtime.InaccessibleAreaException 451
- initialize(long, long, long)**
 - of javax.realtime.PhysicalMemoryTypeFilter 209
- instance()**
 - of javax.realtime.HeapMemory 170
 - of javax.realtime.ImmortalMemory 171
 - of javax.realtime.PriorityInheritance 293
 - of javax.realtime.PriorityScheduler 107
- instance(int)**
 - of javax.realtime.PriorityCeilingEmulation 292
- interrupt()**
 - of javax.realtime.RealtimeThread 36
- interruptAction(AsynchronouslyInterruptedException)**
 - of javax.realtime.Interruptible 423
- Interruptible**
 - of javax.realtime 423
- IO_PAGE**
 - of javax.realtime.PhysicalMemoryManager 201
- isDaemon()**
 - of javax.realtime.AsyncEventHandler 404
- isEmpty()**
 - of javax.realtime.WaitFreeReadQueue 303
 - of javax.realtime.WaitFreeWriteQueue 297
- isEnabled()**
 - of javax.realtime.AsynchronouslyInterruptedException 429
- isFeasible()**
 - of javax.realtime.PriorityScheduler 107
 - of javax.realtime.Scheduler 99
- isFull()**
 - of javax.realtime.WaitFreeReadQueue 303
 - of javax.realtime.WaitFreeWriteQueue 297
- isPresent(long, long)**
 - of javax.realtime.PhysicalMemoryTypeFilter 209
- isRemovable()**
 - of javax.realtime.PhysicalMemoryTypeFilter 210
- isRemovable(long, long)**
 - of javax.realtime.PhysicalMemoryManager 201

isRemoved(long, long)
of `javax.realtime.PhysicalMemoryManager` 201

isRunning()
of `javax.realtime.Timer` 369

J

java.applet - package 461

join()
of `javax.realtime.ScopedMemory` 180

join(HighResolutionTime)
of `javax.realtime.ScopedMemory` 180

joinAndEnter()
of `javax.realtime.ScopedMemory` 181

joinAndEnter(HighResolutionTime)
of `javax.realtime.ScopedMemory` 182

joinAndEnter(Runnable)
of `javax.realtime.ScopedMemory` 184

joinAndEnter(Runnable, HighResolutionTime)
of `javax.realtime.ScopedMemory` 185

L

LITTLE_ENDIAN
of `javax.realtime.RealtimeSystem` 444

LTMemory
of `javax.realtime` 189

LTMemory(long)
of `javax.realtime.LTMemory` 190

LTMemory(long, long)
of `javax.realtime.LTMemory` 191

LTMemory(long, long, Runnable)
of `javax.realtime.LTMemory` 191

LTMemory(long, Runnable)
of `javax.realtime.LTMemory` 192

LTMemory(SizeEstimator)
of `javax.realtime.LTMemory` 192

LTMemory(SizeEstimator, Runnable)
of `javax.realtime.LTMemory` 192

LTMemory(SizeEstimator, SizeEstimator)
of `javax.realtime.LTMemory` 193

LTMemory(SizeEstimator, SizeEstimator, Runnable)
of `javax.realtime.LTMemory` 194

LTPhysicalMemory
of `javax.realtime` 223

LTPhysicalMemory(Object, long)
of `javax.realtime.LTPhysicalMemory` 224

LTPhysicalMemory(Object, long, long)
of `javax.realtime.LTPhysicalMemory` 225

LTPhysicalMemory(Object, long, long, Runnable)
of `javax.realtime.LTPhysicalMemory` 226

LTPhysicalMemory(Object, long, Runnable)
of `javax.realtime.LTPhysicalMemory` 227

LTPhysicalMemory(Object, long, SizeEstimator)
of `javax.realtime.LTPhysicalMemory` 228

LTPhysicalMemory(Object, long, SizeEstimator, Runnable)
of `javax.realtime.LTPhysicalMemory` 229

LTPhysicalMemory(Object, SizeEstimator)
of `javax.realtime.LTPhysicalMemory` 230

LTPhysicalMemory(Object, SizeEstimator, Runnable)
of `javax.realtime.LTPhysicalMemory` 231

M

map()
of `javax.realtime.RawMemoryAccess` 254

map(long)
of `javax.realtime.RawMemoryAccess` 254

map(long, long)
of `javax.realtime.RawMemoryAccess` 255

MAX_PRIORITY
of `javax.realtime.PriorityScheduler` 104

MemoryAccessError
of `javax.realtime` 451

MemoryAccessError()
of `javax.realtime.MemoryAccessError` 452

MemoryAccessError(String)
of `javax.realtime.MemoryAccessError`

- 452
 - MemoryArea**
 - of javax.realtime 163
 - MemoryArea(long)**
 - of javax.realtime.MemoryArea 164
 - MemoryArea(long, Runnable)**
 - of javax.realtime.MemoryArea 164
 - MemoryArea(SizeEstimator)**
 - of javax.realtime.MemoryArea 164
 - MemoryArea(SizeEstimator, Runnable)**
 - of javax.realtime.MemoryArea 165
 - memoryConsumed()**
 - of javax.realtime.MemoryArea 167
 - MemoryInUseException**
 - of javax.realtime 452
 - MemoryInUseException()**
 - of javax.realtime.MemoryInUseException 452
 - MemoryInUseException(String)**
 - of javax.realtime.MemoryInUseException 452
 - MemoryParameters**
 - of javax.realtime 275
 - MemoryParameters(long, long)**
 - of javax.realtime.MemoryParameters 276
 - MemoryParameters(long, long, long)**
 - of javax.realtime.MemoryParameters 276
 - memoryRemaining()**
 - of javax.realtime.MemoryArea 167
 - MemoryScopeException**
 - of javax.realtime 453
 - MemoryScopeException()**
 - of javax.realtime.MemoryScopeException 453
 - MemoryScopeException(String)**
 - of javax.realtime.MemoryScopeException 453
 - MemoryTypeConflictException**
 - of javax.realtime 453
 - MemoryTypeConflictException()**
 - of javax.realtime.MemoryTypeConflictException 454
 - MemoryTypeConflictException(String)**
 - of javax.realtime.MemoryTypeConflictException 454
 - MIN_PRIORITY**
 - of javax.realtime.PriorityScheduler 104
 - mitViolationExcept**
 - of javax.realtime.SporadicParameters
 - 138
 - MITViolationException**
 - of javax.realtime 454
 - MITViolationException()**
 - of javax.realtime.MITViolationException 455
 - MITViolationException(String)**
 - of javax.realtime.MITViolationException 455
 - mitViolationIgnore**
 - of javax.realtime.SporadicParameters 139
 - mitViolationReplace**
 - of javax.realtime.SporadicParameters 139
 - mitViolationSave**
 - of javax.realtime.SporadicParameters 139
 - MonitorControl**
 - of javax.realtime 288
 - MonitorControl()**
 - of javax.realtime.MonitorControl 288
- N**
- newArray(Class, int)**
 - of javax.realtime.MemoryArea 167
 - of javax.realtime.ScopedMemory 186
 - newInstance(Class)**
 - of javax.realtime.MemoryArea 168
 - of javax.realtime.ScopedMemory 187
 - newInstance(Constructor, Object[])**
 - of javax.realtime.MemoryArea 168
 - of javax.realtime.ScopedMemory 188
 - NO_MAX**
 - of javax.realtime.MemoryParameters 275
 - NoHeapRealtimeThread**
 - of javax.realtime 55
 - NoHeapRealtimeThread(SchedulingParameters, MemoryArea)**
 - of javax.realtime.NoHeapRealtimeThread 56
 - NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryArea)**
 - of javax.realtime.NoHeapRealtimeThread 56
 - NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, Memo-**

ryParameters, MemoryArea,
 ProcessingGroupParameters, Run-
 nable)
 of javax.realtime.NoHeapRealtimeThread
 57
nonBlockingRead()
 of javax.realtime.WaitFreeDequeue 308
nonBlockingWrite(Object)
 of javax.realtime.WaitFreeDequeue 309
O
OffsetOutOfBoundsException
 of javax.realtime 455
OffsetOutOfBoundsException()
 of javax.realtime.OffsetOutOfBoundsEx-
 ception 455
OffsetOutOfBoundsException(String)
 of javax.realtime.OffsetOutOfBoundsEx-
 ception 455
OneShotTimer
 of javax.realtime 371
**OneShotTimer(HighResolutionTime, Async-
 cEventHandler)**
 of javax.realtime.OneShotTimer 372
**OneShotTimer(HighResolutionTime, Clock,
 AsyncEventHandler)**
 of javax.realtime.OneShotTimer 372
onInsertion(long, long, AsyncEvent)
 of javax.realtime.PhysicalMemoryMan-
 ager 202
 of javax.realtime.PhysicalMemoryType-
 Filter 210
onInsertion(long, long, AsyncEventHandler)
 of javax.realtime.PhysicalMemoryMan-
 ager 203
 of javax.realtime.PhysicalMemoryType-
 Filter 211
onRemoval(long, long, AsyncEvent)
 of javax.realtime.PhysicalMemoryMan-
 ager 203
 of javax.realtime.PhysicalMemoryType-
 Filter 211
onRemoval(long, long, AsyncEventHandler)
 of javax.realtime.PhysicalMemoryMan-
 ager 204
 of javax.realtime.PhysicalMemoryType-
 Filter 212

P

PeriodicParameters
 of javax.realtime 123
**PeriodicParameters(HighResolutionTime,
 RelativeTime)**
 of javax.realtime.PeriodicParameters 125
**PeriodicParameters(HighResolutionTime,
 RelativeTime, RelativeTime, Rela-
 tiveTime, AsyncEventHandler,
 AsyncEventHandler)**
 of javax.realtime.PeriodicParameters 126
PeriodicParameters(RelativeTime)
 of javax.realtime.PeriodicParameters 127
PeriodicTimer
 of javax.realtime 373
**PeriodicTimer(HighResolutionTime, Rela-
 tiveTime, AsyncEventHandler)**
 of javax.realtime.PeriodicTimer 374
**PeriodicTimer(HighResolutionTime, Rela-
 tiveTime, Clock, Async-
 cEventHandler)**
 of javax.realtime.PeriodicTimer 375
PhysicalMemoryManager
 of javax.realtime 199
PhysicalMemoryTypeFilter
 of javax.realtime 207
POSIXSignalHandler
 of javax.realtime 434
PriorityCeilingEmulation
 of javax.realtime 290
PriorityInheritance
 of javax.realtime 293
PriorityParameters
 of javax.realtime 113
PriorityParameters(int)
 of javax.realtime.PriorityParameters 114
PriorityScheduler
 of javax.realtime 103
PriorityScheduler()
 of javax.realtime.PriorityScheduler 104
ProcessingGroupParameters
 of javax.realtime 144
**ProcessingGroupParameters(HighResolu-
 tionTime, RelativeTime, Rela-
 tiveTime, RelativeTime,
 AsyncEventHandler, Async-
 cEventHandler)**
 of javax.realtime.ProcessingGroupParam-

- eters 146
- propagate()**
 - of javax.realtime.AsynchronouslyInterruptedException 429
- R**
- RationalTime**
 - of javax.realtime 344
- RationalTime(int)**
 - of javax.realtime.RationalTime 344
- RationalTime(int, long, int)**
 - of javax.realtime.RationalTime 344
- RationalTime(int, RelativeTime)**
 - of javax.realtime.RationalTime 345
- RawMemoryAccess**
 - of javax.realtime 241
- RawMemoryAccess(Object, long)**
 - of javax.realtime.RawMemoryAccess 244
- RawMemoryAccess(Object, long, long)**
 - of javax.realtime.RawMemoryAccess 245
- RawMemoryFloatAccess**
 - of javax.realtime 264
- RawMemoryFloatAccess(Object, long)**
 - of javax.realtime.RawMemoryFloatAccess 265
- RawMemoryFloatAccess(Object, long, long)**
 - of javax.realtime.RawMemoryFloatAccess 266
- read()**
 - of javax.realtime.WaitFreeReadQueue 303
 - of javax.realtime.WaitFreeWriteQueue 297
- RealtimeSecurity**
 - of javax.realtime 441
- RealtimeSecurity()**
 - of javax.realtime.RealtimeSecurity 441
- RealtimeSystem**
 - of javax.realtime 443
- RealtimeThread**
 - of javax.realtime 29
- RealtimeThread()**
 - of javax.realtime.RealtimeThread 30
- RealtimeThread(SchedulingParameters)**
 - of javax.realtime.RealtimeThread 30
- RealtimeThread(SchedulingParameters, ReleaseParameters)**
 - of javax.realtime.RealtimeThread 30
- RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)**
 - of javax.realtime.RealtimeThread 31
- registerFilter(Object, PhysicalMemoryTypeFilter)**
 - of javax.realtime.PhysicalMemoryManager 205
- relative(Clock)**
 - of javax.realtime.AbsoluteTime 330
 - of javax.realtime.HighResolutionTime 319
 - of javax.realtime.RelativeTime 341
- relative(Clock, RelativeTime)**
 - of javax.realtime.AbsoluteTime 330
 - of javax.realtime.HighResolutionTime 319
 - of javax.realtime.RelativeTime 341
- RelativeTime**
 - of javax.realtime 334
- RelativeTime()**
 - of javax.realtime.RelativeTime 335
- RelativeTime(Clock)**
 - of javax.realtime.RelativeTime 335
- RelativeTime(long, int)**
 - of javax.realtime.RelativeTime 335
- RelativeTime(long, int, Clock)**
 - of javax.realtime.RelativeTime 335
- RelativeTime(RelativeTime)**
 - of javax.realtime.RelativeTime 336
- RelativeTime(RelativeTime, Clock)**
 - of javax.realtime.RelativeTime 336
- ReleaseParameters**
 - of javax.realtime 116
- ReleaseParameters()**
 - of javax.realtime.ReleaseParameters 118
- ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)**
 - of javax.realtime.ReleaseParameters 118
- removeFilter(Object)**
 - of javax.realtime.PhysicalMemoryManager 205
- removeFromFeasibility()**
 - of javax.realtime.AsyncEventHandler

- 404
 - of javax.realtime.RealtimeThread 36
 - of javax.realtime.Schedulable 82
 - removeFromFeasibility(Schedulable)**
 - of javax.realtime.PriorityScheduler 108
 - of javax.realtime.Scheduler 99
 - removeHandler(AsyncEventHandler)**
 - of javax.realtime.AsyncEvent 393
 - of javax.realtime.Timer 369
 - removeHandler(int, AsyncEventHandler)**
 - of javax.realtime.POSIXSignalHandler 440
 - reschedule(HighResolutionTime)**
 - of javax.realtime.Timer 369
 - reserve(Class, int)**
 - of javax.realtime.SizeEstimator 172
 - reserve(SizeEstimator)**
 - of javax.realtime.SizeEstimator 173
 - reserve(SizeEstimator, int)**
 - of javax.realtime.SizeEstimator 173
 - reserveArray(int)**
 - of javax.realtime.SizeEstimator 173
 - reserveArray(int, Class)**
 - of javax.realtime.SizeEstimator 174
 - resetTime(HighResolutionTime)**
 - of javax.realtime.Timed 431
 - ResourceLimitError**
 - of javax.realtime 459
 - ResourceLimitError()**
 - of javax.realtime.ResourceLimitError 459
 - ResourceLimitError(String)**
 - of javax.realtime.ResourceLimitError 459
 - run()**
 - of javax.realtime.AsyncEventHandler 404
 - run(AsynchronouslyInterruptedException)**
 - of javax.realtime.Interruptible 423
- S**
- Schedulable**
 - of javax.realtime 81
 - schedulePeriodic()**
 - of javax.realtime.RealtimeThread 37
 - Scheduler**
 - of javax.realtime 97
 - Scheduler()**
 - of javax.realtime.Scheduler 97
 - SchedulingParameters**
 - of javax.realtime 112
 - SchedulingParameters()**
 - of javax.realtime.SchedulingParameters 113
 - ScopedCycleException**
 - of javax.realtime 456
 - ScopedCycleException()**
 - of javax.realtime.ScopedCycleException 456
 - ScopedCycleException(String)**
 - of javax.realtime.ScopedCycleException 456
 - ScopedMemory**
 - of javax.realtime 174
 - ScopedMemory(long)**
 - of javax.realtime.ScopedMemory 175
 - ScopedMemory(long, Runnable)**
 - of javax.realtime.ScopedMemory 175
 - ScopedMemory(SizeEstimator)**
 - of javax.realtime.ScopedMemory 176
 - ScopedMemory(SizeEstimator, Runnable)**
 - of javax.realtime.ScopedMemory 176
 - set(Date)**
 - of javax.realtime.AbsoluteTime 331
 - set(HighResolutionTime)**
 - of javax.realtime.HighResolutionTime 320
 - set(long)**
 - of javax.realtime.HighResolutionTime 320
 - set(long, int)**
 - of javax.realtime.HighResolutionTime 321
 - of javax.realtime.RationalTime 346
 - setAllocationRate(long)**
 - of javax.realtime.MemoryParameters 278
 - setAllocationRateIfFeasible(long)**
 - of javax.realtime.MemoryParameters 278
 - setArrivalTimeQueueOverflowBehavior(String)**
 - of javax.realtime.AperiodicParameters 135
 - setByte(long, byte)**
 - of javax.realtime.RawMemoryAccess 256

- setBytes(long, byte[], int, int)**
 - of javax.realtime.RawMemoryAccess 256
- setCost(RelativeTime)**
 - of javax.realtime.ProcessingGroupParameters 148
 - of javax.realtime.ReleaseParameters 120
- setCostOverrunHandler(AsyncEventHandler)**
 - of javax.realtime.ProcessingGroupParameters 149
 - of javax.realtime.ReleaseParameters 121
- setDaemon(boolean)**
 - of javax.realtime.AsyncEventHandler 405
- setDeadline(RelativeTime)**
 - of javax.realtime.AperiodicParameters 135
 - of javax.realtime.PeriodicParameters 128
 - of javax.realtime.ProcessingGroupParameters 149
 - of javax.realtime.ReleaseParameters 121
- setDeadlineMissHandler(AsyncEventHandler)**
 - of javax.realtime.ProcessingGroupParameters 149
 - of javax.realtime.ReleaseParameters 122
- setDefaultScheduler(Scheduler)**
 - of javax.realtime.Scheduler 99
- setDouble(long, double)**
 - of javax.realtime.RawMemoryFloatAccess 271
- setDoubles(long, double[], int, int)**
 - of javax.realtime.RawMemoryFloatAccess 272
- setFloat(long, float)**
 - of javax.realtime.RawMemoryFloatAccess 273
- setFloats(long, float[], int, int)**
 - of javax.realtime.RawMemoryFloatAccess 274
- setFrequency(int)**
 - of javax.realtime.RationalTime 347
- setHandler(AsyncEventHandler)**
 - of javax.realtime.AsyncEvent 394
 - of javax.realtime.Timer 370
- setHandler(int, AsyncEventHandler)**
 - of javax.realtime.POSIXSignalHandler 440
- setIfFeasible(RelativeTime, RelativeTime)**
 - of javax.realtime.AperiodicParameters 136
 - of javax.realtime.ReleaseParameters 122
 - of javax.realtime.SporadicParameters 141
- setIfFeasible(RelativeTime, RelativeTime, RelativeTime)**
 - of javax.realtime.PeriodicParameters 128
 - of javax.realtime.ProcessingGroupParameters 150
 - of javax.realtime.SporadicParameters 142
- setIfFeasible(ReleaseParameters, MemoryParameters)**
 - of javax.realtime.AsyncEventHandler 406
 - of javax.realtime.RealtimeThread 37
 - of javax.realtime.Schedulable 83
- setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)**
 - of javax.realtime.AsyncEventHandler 407
 - of javax.realtime.RealtimeThread 38
 - of javax.realtime.Schedulable 84
- setIfFeasible(ReleaseParameters, ProcessingGroupParameters)**
 - of javax.realtime.AsyncEventHandler 408
 - of javax.realtime.RealtimeThread 39
 - of javax.realtime.Schedulable 85
- setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters)**
 - of javax.realtime.PriorityScheduler 109
 - of javax.realtime.Scheduler 100
- setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)**
 - of javax.realtime.PriorityScheduler 110
 - of javax.realtime.Scheduler 101
- setIfFeasible(Schedulable, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)**
 - of javax.realtime.PriorityScheduler 111
 - of javax.realtime.Scheduler 102
- setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters,**

- ters)
 - of javax.realtime.AsyncEventHandler 409
 - of javax.realtime.RealtimeThread 40
 - of javax.realtime.Schedulable 86
- setIfFeasible(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)**
 - of javax.realtime.AsyncEventHandler 410
 - of javax.realtime.RealtimeThread 42
 - of javax.realtime.Schedulable 88
- setImportance(int)**
 - of javax.realtime.ImportanceParameters 116
- setInitialArrivalTimeQueueLength(int)**
 - of javax.realtime.AperiodicParameters 137
- setInt(long, int)**
 - of javax.realtime.RawMemoryAccess 258
- setInterval(RelativeTime)**
 - of javax.realtime.PeriodicTimer 378
- setInts(long, int[], int, int)**
 - of javax.realtime.RawMemoryAccess 258
- setLong(long, long)**
 - of javax.realtime.RawMemoryAccess 259
- setLongs(long, long[], int, int)**
 - of javax.realtime.RawMemoryAccess 260
- setMaxImmortalIfFeasible(long)**
 - of javax.realtime.MemoryParameters 279
- setMaximumConcurrentLocks(int)**
 - of javax.realtime.RealtimeSystem 445
- setMaximumConcurrentLocks(int, boolean)**
 - of javax.realtime.RealtimeSystem 445
- setMaxMemoryAreaIfFeasible(long)**
 - of javax.realtime.MemoryParameters 279
- setMemoryParameters(MemoryParameters)**
 - of javax.realtime.AsyncEventHandler 412
 - of javax.realtime.RealtimeThread 43
 - of javax.realtime.Schedulable 89
- setMemoryParametersIfFeasible(MemoryParameters)**
 - of javax.realtime.AsyncEventHandler 413
 - of javax.realtime.RealtimeThread 44
 - of javax.realtime.Schedulable 90
- setMinimumInterarrival(RelativeTime)**
 - of javax.realtime.SporadicParameters 143
- setMitViolationBehavior(String)**
 - of javax.realtime.SporadicParameters 143
- setMonitorControl(MonitorControl)**
 - of javax.realtime.MonitorControl 289
- setMonitorControl(Object, MonitorControl)**
 - of javax.realtime.MonitorControl 289
- setPeriod(RelativeTime)**
 - of javax.realtime.PeriodicParameters 129
 - of javax.realtime.ProcessingGroupParameters 150
- setPortal(Object)**
 - of javax.realtime.ScopedMemory 189
- setPriority(int)**
 - of javax.realtime.PriorityParameters 114
- setProcessingGroupParameters(ProcessingGroupParameters)**
 - of javax.realtime.AsyncEventHandler 413
 - of javax.realtime.RealtimeThread 45
 - of javax.realtime.Schedulable 90
- setProcessingGroupParametersIfFeasible(ProcessingGroupParameters)**
 - of javax.realtime.AsyncEventHandler 414
 - of javax.realtime.RealtimeThread 45
 - of javax.realtime.Schedulable 91
- setReleaseParameters(ReleaseParameters)**
 - of javax.realtime.AsyncEventHandler 415
 - of javax.realtime.RealtimeThread 46
 - of javax.realtime.Schedulable 92
- setReleaseParametersIfFeasible(ReleaseParameters)**
 - of javax.realtime.AsyncEventHandler 416
 - of javax.realtime.RealtimeThread 47
 - of javax.realtime.Schedulable 93
- setResolution(RelativeTime)**
 - of javax.realtime.Clock 356
- setScheduler(Scheduler)**
 - of javax.realtime.AsyncEventHandler 417

- of `javax.realtime.RealtimeThread` 48
 - of `javax.realtime.Schedulable` 94
- setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)**
 - of `javax.realtime.AsyncEventHandler` 418
 - of `javax.realtime.RealtimeThread` 49
 - of `javax.realtime.Schedulable` 94
- setSchedulingParameters(SchedulingParameters)**
 - of `javax.realtime.AsyncEventHandler` 419
 - of `javax.realtime.RealtimeThread` 50
 - of `javax.realtime.Schedulable` 95
- setSchedulingParametersIfFeasible(SchedulingParameters)**
 - of `javax.realtime.AsyncEventHandler` 420
 - of `javax.realtime.RealtimeThread` 51
 - of `javax.realtime.Schedulable` 96
- setSecurityManager(RealtimeSecurity)**
 - of `javax.realtime.RealtimeSystem` 446
- setShort(long, short)**
 - of `javax.realtime.RawMemoryAccess` 261
- setShorts(long, short[], int, int)**
 - of `javax.realtime.RawMemoryAccess` 262
- setStart(HighResolutionTime)**
 - of `javax.realtime.PeriodicParameters` 130
 - of `javax.realtime.ProcessingGroupParameters` 151
- SHARED**
 - of `javax.realtime.PhysicalMemoryManager` 201
- SIGABRT**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGALRM**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGBUS**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGCANCEL**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGCHLD**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGCLD**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGCONT**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGEMT**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGFPE**
 - of `javax.realtime.POSIXSignalHandler` 435
- SIGFREEZE**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGHUP**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGILL**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGINT**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGIO**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGIOT**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGKILL**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGLOST**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGLWP**
 - of `javax.realtime.POSIXSignalHandler` 436
- SIGPIPE**
 - of `javax.realtime.POSIXSignalHandler` 437
- SIGPOLL**
 - of `javax.realtime.POSIXSignalHandler` 437

SIGPROF

of `javax.realtime.POSIXSignalHandler`
437

SIGPWR

of `javax.realtime.POSIXSignalHandler`
437

SIGQUIT

of `javax.realtime.POSIXSignalHandler`
437

SIGSEGV

of `javax.realtime.POSIXSignalHandler`
437

SIGSTOP

of `javax.realtime.POSIXSignalHandler`
437

SIGSYS

of `javax.realtime.POSIXSignalHandler`
437

SIGTERM

of `javax.realtime.POSIXSignalHandler`
438

SIGTHAW

of `javax.realtime.POSIXSignalHandler`
438

SIGTRAP

of `javax.realtime.POSIXSignalHandler`
438

SIGTSTP

of `javax.realtime.POSIXSignalHandler`
438

SIGTTIN

of `javax.realtime.POSIXSignalHandler`
438

SIGTTOU

of `javax.realtime.POSIXSignalHandler`
438

SIGURG

of `javax.realtime.POSIXSignalHandler`
438

SIGUSR1

of `javax.realtime.POSIXSignalHandler`
438

SIGUSR2

of `javax.realtime.POSIXSignalHandler`
439

SIGVTALRM

of `javax.realtime.POSIXSignalHandler`
439

SIGWAITING

of `javax.realtime.POSIXSignalHandler`
439

SIGWINCH

of `javax.realtime.POSIXSignalHandler`
439

SIGXCPU

of `javax.realtime.POSIXSignalHandler`
439

SIGXFSZ

of `javax.realtime.POSIXSignalHandler`
439

size()

of `javax.realtime.MemoryArea` 169
of `javax.realtime.WaitFreeReadQueue`
304
of `javax.realtime.WaitFreeWriteQueue`
298

SizeEstimator

of `javax.realtime` 171

SizeEstimator()

of `javax.realtime.SizeEstimator` 172

SizeOutOfBoundsException

of `javax.realtime` 456

SizeOutOfBoundsException()

of `javax.realtime.SizeOutOfBoundsException`
457

SizeOutOfBoundsException(String)

of `javax.realtime.SizeOutOfBoundsException`
457

sleep(Clock, HighResolutionTime)

of `javax.realtime.RealtimeThread` 52

sleep(HighResolutionTime)

of `javax.realtime.RealtimeThread` 53

SporadicParameters

of `javax.realtime` 137

SporadicParameters(RelativeTime)

of `javax.realtime.SporadicParameters`
139

SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

of `javax.realtime.SporadicParameters`
140

start()

of `javax.realtime.NoHeapRealtimeThread`
58
of `javax.realtime.RealtimeThread` 54

- of `javax.realtime.Timer` 370
- start(boolean)**
 - of `javax.realtime.Timer` 371
- stop()**
 - of `javax.realtime.Timer` 371
- subtract(AbsoluteTime)**
 - of `javax.realtime.AbsoluteTime` 331
- subtract(AbsoluteTime, RelativeTime)**
 - of `javax.realtime.AbsoluteTime` 332
- subtract(RelativeTime)**
 - of `javax.realtime.AbsoluteTime` 332
 - of `javax.realtime.RelativeTime` 342
- subtract(RelativeTime, AbsoluteTime)**
 - of `javax.realtime.AbsoluteTime` 333
- subtract(RelativeTime, RelativeTime)**
 - of `javax.realtime.RelativeTime` 342

T

- ThrowBoundaryError**
 - of `javax.realtime` 457
- ThrowBoundaryError()**
 - of `javax.realtime.ThrowBoundaryError` 457
- ThrowBoundaryError(String)**
 - of `javax.realtime.ThrowBoundaryError` 457
- Timed**
 - of `javax.realtime` 430
- Timed(HighResolutionTime)**
 - of `javax.realtime.Timed` 430
- Timer**
 - of `javax.realtime` 356
- Timer(HighResolutionTime, Clock, AsyncEventHandler)**
 - of `javax.realtime.Timer` 364
- toString()**
 - of `javax.realtime.AbsoluteTime` 334
 - of `javax.realtime.ImportanceParameters` 116
 - of `javax.realtime.LTMemory` 194
 - of `javax.realtime.LTPhysicalMemory` 232
 - of `javax.realtime.PriorityParameters` 115
 - of `javax.realtime.RationalTime` 347
 - of `javax.realtime.RelativeTime` 343
 - of `javax.realtime.ScopedMemory` 189
 - of `javax.realtime.VTMemory` 199
 - of `javax.realtime.VTPhysicalMemory`

241

U

- unbindTo(String)**
 - of `javax.realtime.AsyncEvent` 394
- UnknownHappeningException**
 - of `javax.realtime` 458
- UnknownHappeningException()**
 - of `javax.realtime.UnknownHappeningException` 459
- UnknownHappeningException(String)**
 - of `javax.realtime.UnknownHappeningException` 459
- unmap()**
 - of `javax.realtime.RawMemoryAccess` 263
- unregisterInsertionEvent(long, long, AsyncEvent)**
 - of `javax.realtime.PhysicalMemoryManager` 206
 - of `javax.realtime.PhysicalMemoryTypeFilter` 212
- unregisterRemovalEvent(long, long, AsyncEvent)**
 - of `javax.realtime.PhysicalMemoryManager` 206
 - of `javax.realtime.PhysicalMemoryTypeFilter` 213
- UnsupportedPhysicalMemoryException**
 - of `javax.realtime` 458
- UnsupportedPhysicalMemoryException()**
 - of `javax.realtime.UnsupportedPhysicalMemoryException` 458
- UnsupportedPhysicalMemoryException(String)**
 - of `javax.realtime.UnsupportedPhysicalMemoryException` 458

V

- vFind(long, long)**
 - of `javax.realtime.PhysicalMemoryTypeFilter` 214
- VTMemory**
 - of `javax.realtime` 195
- VTMemory(long)**
 - of `javax.realtime.VTMemory` 195

- VTMemory(long, long)**
 - of javax.realtime.VTMemory 195
 - VTMemory(long, long, Runnable)**
 - of javax.realtime.VTMemory 196
 - VTMemory(long, Runnable)**
 - of javax.realtime.VTMemory 196
 - VTMemory(SizeEstimator)**
 - of javax.realtime.VTMemory 197
 - VTMemory(SizeEstimator, Runnable)**
 - of javax.realtime.VTMemory 197
 - VTMemory(SizeEstimator, SizeEstimator)**
 - of javax.realtime.VTMemory 198
 - VTMemory(SizeEstimator, SizeEstimator, Runnable)**
 - of javax.realtime.VTMemory 198
 - VTPhysicalMemory**
 - of javax.realtime 232
 - VTPhysicalMemory(Object, long)**
 - of javax.realtime.VTPhysicalMemory 233
 - VTPhysicalMemory(Object, long, long)**
 - of javax.realtime.VTPhysicalMemory 234
 - VTPhysicalMemory(Object, long, long, Runnable)**
 - of javax.realtime.VTPhysicalMemory 235
 - VTPhysicalMemory(Object, long, Runnable)**
 - of javax.realtime.VTPhysicalMemory 236
 - VTPhysicalMemory(Object, long, SizeEstimator)**
 - of javax.realtime.VTPhysicalMemory 237
 - VTPhysicalMemory(Object, long, SizeEstimator, Runnable)**
 - of javax.realtime.VTPhysicalMemory 238
 - VTPhysicalMemory(Object, SizeEstimator)**
 - of javax.realtime.VTPhysicalMemory 239
 - VTPhysicalMemory(Object, SizeEstimator, Runnable)**
 - of javax.realtime.VTPhysicalMemory 240
- W**
- waitForData()**
 - of javax.realtime.WaitFreeReadQueue 304
 - waitForNextPeriod()**
 - of javax.realtime.RealtimeThread 54
 - waitForNextPeriodInterruptible()**
 - of javax.realtime.RealtimeThread 54
 - waitForObject(Object, HighResolutionTime)**
 - of javax.realtime.HighResolutionTime 321
 - WaitFreeDequeue**
 - of javax.realtime 305
 - WaitFreeDequeue(Runnable, Runnable, int, MemoryArea)**
 - of javax.realtime.WaitFreeDequeue 306
 - WaitFreeReadQueue**
 - of javax.realtime 299
 - WaitFreeReadQueue(int, boolean)**
 - of javax.realtime.WaitFreeReadQueue 300
 - WaitFreeReadQueue(int, MemoryArea, boolean)**
 - of javax.realtime.WaitFreeReadQueue 300
 - WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea)**
 - of javax.realtime.WaitFreeReadQueue 301
 - WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea, boolean)**
 - of javax.realtime.WaitFreeReadQueue 302
 - WaitFreeWriteQueue**
 - of javax.realtime 294
 - WaitFreeWriteQueue(int)**
 - of javax.realtime.WaitFreeWriteQueue 295
 - WaitFreeWriteQueue(int, MemoryArea)**
 - of javax.realtime.WaitFreeWriteQueue 295
 - WaitFreeWriteQueue(Runnable, Runnable, int, MemoryArea)**
 - of javax.realtime.WaitFreeWriteQueue 295
 - write(Object)**
 - of javax.realtime.WaitFreeReadQueue

[304](#)

of javax.realtime.WaitFreeWriteQueue

[298](#)

Unofficial

Unofficial