(DRAFT) The Real-Time Specification for Java™

The Real-Time for Java Expert Group

http://www.rtj.org

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

ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 00-132774

ISBN 0-201-70323-8
Text printed on recycled paper.

1 2 3 4 5 6 7 8 9 10-MA-04 03 02 01 00
First printing, June 2000

To Vicki, who has been committed to our effort from the beginning — GB

To Deb, Abz, and Dan, for making it all worthwhile — BB

To Ken Kaplan and my family, who allowed me the
time and resources for this work — PD

To Linda, who has always been a true friend, cared for my home in my absences,
welcomed me at the airport and generally shown patience and consideration — SF

To Judy, Kelsey, and Kate, who gave me the
Love and Time to work on this book — JG

To Debbie, Sam, and Anna, who endured my frequent absences, and general
absentmindedness, during the writing of this book — DH

To my daughters Christine, Heather, and Victoria, and especially to my wife Terry,
who all put up with my strange working hours — MT

To my mother Maria, brother Luigi, sister-in-law Claude, and nephew Nicola — RB

To my wife Kathy for her unflagging support throughout this effort, and her patience with
the time required to complete this work — DL

To my mother Donna and my father Jerry, who put up with me all these years, and my
brother Kenneth who introduced me to computers in the first place — SR

To my wife Sohini, for her love and understanding — PS

To my wife Chelo, for her love, support and understanding in this journey; and to my
daughters Ana and Sofia, the light of the journey — DdN

To the Stanford Inn-by-the-Sea, the Chicago Hilton, and the Chateau Laurier for
providing space for a bunch of geeks to hang out; and to the Beaver Tail vendors by the
Rideau Canal for providing a yummy distraction.

Contents

Caveat

This edition of *The Real-Time Specification for Java*™ (RTSJ) is **preliminary**. It is being developed under the Java Community Process (http://java.sun.com/aboutJava/communityprocess) (http://java.sun.com/aboutJava/communityprocess). It will not be considered final until after the completion of the reference implementation. The experience gained from that implementation may necessitate changes to the specification. Status information on the specification may be obtained from the web site maintained by the expert group, **http://www.rtj.org (http://www.rtj.org)**, along with updates and samples.

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. If we refer to other heaps, such as the heap used by the C language runtime or the operating system's heap, we will explicitly state which heap.

Throughout the RTSJ we will use the term *Thread* to refer to the class `Thread` in *The Java Language Specification* and *thread* to refer to a sequence of instructions or to an instance of the class `Thread`. The context of uses of *thread* should be sufficient to distinguish between the two meanings. We will be explicit where we think necessary.

In order to get this published and in your hands, we made some compromises in copyediting and proofreading for this first edition. It is our intention to provide this book for you to begin designing real-time applications with this specification. Please send any and all comments to: comments@rtj.org.

Authors

Greg Bollella, a Senior Staff Engineer for Sun Microsystems Laboratories, is Principle Investigator for Real-Time Java. Previously, while a Senior Architect at IBM, Greg created and led the Real-Time for Java Expert Group under The Java Community Process. Greg has also designed and implemented communications protocols for 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, Senior Scientist at Microware Systems Corporation, has designed, coded, and analyzed system software for real-time systems for more than ten years with particular emphasis on real-time performance issues. As part of Microware's Java team, Peter has been involved with the Java Virtual Machine since early 1997.

Steve Furr currently works for QNX Software Systems, where he is 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.

Foreword

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

Preface

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). aJile 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 processess 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/aJile
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 Sueltz, 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 Haggar 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 (see the Colophon for more details) 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 in the book by including on each occurrence of a name the page number of its definition as a trailing subscript. Let us know if this is a useful feature (comments@rtj.org).

CHAPTER 1

Introduction

This book is a preliminary release of *The Real-Time Specification for Java*™ (RTSJ). The final version will be available with the release of the reference implementation.

The Real-Time for Java Expert Group (RTJEG), convened under the Java Community Process and JSR-000001, has been 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 our 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 we identified as requiring enhancements to accomplish our goal.

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 tradeoffs; 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, tradeoffs 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 ~~exact~~ 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 accomodate 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 needs to share serializable~~ 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 ~~inverison~~ 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 sytems 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.

CHAPTER 2

Design

The RTSJ comprises ~~eight~~ seven areas of extended semantics. This chapter explains each in fair detail. Further detail, exact requirements, and rationale are given in the opening section of each relevant chapter. The ~~eight~~ 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 ~~and exceptions~~—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.* Any instance of any class implementing the interface `Schedulable` is a schedulable object and its scheduling and dispatching will be managed by the instance of `Scheduler` to which it holds a reference. The RTSJ requires three classes that are schedulable objects; `RealtimeThread, NoHeapRealtimeThread`, and `AsyncEventHandler.`

By *timely execution of threads,* we mean 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 threads (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~~ 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 thread; 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 to determine feasibility. ~~The feasibility algorithms assume that the rate-monotonic priority assignment algorithm has been used to assign priorities. The RTSJ does not require that implementations check that such a priority assignment is correct. If, of course, the assignment is incorrect the feasibility analysis will be meaningless (note however, that this is no different than the vast majority of real-time operating systems and kernels in use today).~~ For the base scheduler the feasibility methods may always return an indication that the system is feasible. This is allowed because the application controls the assignment of priority and the RTSJ does not check assignment. The RTSJ expects that the base scheduler may be subclassed in particular implementations (e.g., an RMA or 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~~s~~ (thread create-time or set (reset))~~, later~~ later, 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 `PriorityParameter` 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. Programming to the Java platform may result in code much closer toward the goal of reusing software written once but able to execute on many different computing platforms (known as Write Once, Run Anywhere) and that the above flexibility stands in opposition to that goal, *The Real-Time Specification for Java* also specifies a particular scheduling algorithm and semantic ~~changes to~~ constraints on the JVM that support predictable execution and must be available on all implementations of the RTSJ. The initial default and required scheduling algorithm is fixed-priority preemptive with at least 28 unique priority levels and will be represented in all implementations by the `PriorityScheduler` subclass of `Scheduler`.

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 for dealing with a class of objects that have a lifetime defined by syntactic scope (cf, the lifetime of objects on the heap).~~ Scoped memory provides a mechanism, more general than stack allocated objects, for managing objects that have a lifetime defined by scope (cf, the life-time of objects on the heap).

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, once allocated, exist until the end of the application~~, i.e., the objects are immortal~~.
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 `RealtimeThread` which will effectively enter the scope before it executes the thread's `run()` method.

Every scoped memory area effectively maintains a count of the number of external references to that memory area. The reference count for a `ScopedMemory` area is increased by entering a new scope through the `enter()` method of `MemoryArea`, by the creation of a `RealtimeThread` using the particular `ScopedMemory` area, or by the opening of an inner scope. The reference count for a `ScopedMemory` area is decreased when returning from the `enter()` method, when the `RealtimeThread` using the `ScopedMemory` exits, 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 is executed to completion. The scope cannot be reused until finalization is complete and the RTSJ requires that the finalizers execute to completion before the next use (calling `enter()` or in a constructor) of the scoped memory area.

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 unusual 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 enclosing 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 threads in an application. Objects allocated in `ImmortalMemory` are freed only when the Java runtime environment terminates, and are never subject to garbage collection or movement.

Budgeted Allocation

The RTSJ also provides limited support for providing memory allocation budgets for threads using memory areas. Maximum memory area consumption and maximum allocation rates for individual real-time threads may be specified when the thread is 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 waiting to acquire a resource must be released in execution eligibility order. This applies to the processor as well as to synchronized blocks. If threads with the same execution eligibility are possible under the active scheduling policy, such threads 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 stat-

ing 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. The highest locker protocol is also a well-known algorithm in the literature, and it has the following effect:

- With this policy, a monitor is given a *priority ceiling* when it is created, which is the highest priority of any thread that could attempt to enter the monitor.
- As soon as a thread enters synchronized code, its priority is raised to the monitor's ceiling priority, thus ensuring mutually exclusive access to the code since it will not be preempted by any thread that could possibly attempt to enter the same monitor.
- If, through programming error, a thread has a higher priority than the ceiling of the monitor it is attempting to enter, then an exception is thrown.

One needs to consider the design point given above, the two new thread types, `RealtimeThread` and `NoHeapRealtimeThread`, and regular Java threads and the possible issues that could arise when a `NoHeapRealtimeThread` and a regular Java thread attempt to synchronize on the same object. `NoHeapRealtimeThreads` mayhave an ~~implicit~~ execution eligibility ~~that must be~~ higher than that of the garbage collector. This is fundamental to the RTSJ. However, given that regular Java threads may never have an execution eligibility higher than the garbage collector, no known priority inversion avoidance algorithm can be correctly implemented when the shared object is shared between a regular Java thread and a `NoHeapRealtimeThread` because the algorithm may not raise the priority of the regular Java thread higher than the garbage collector. Some mechanism other than the synchronized keyword is needed to ensure non-blocking, protected access to objects shared between regular Java threads and NoHeapRealtimeThreads.

Note that if the RTSJ requires that the execution of NoHeapRealtimeThreads must not be delayed by the execution of the garbage collector it is impossible for a NoHeapRealtimeThread to synchronize, in the classic sense, on an object accessed by regular Java threads. The RTSJ provides three wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and NoHeapRealtimeThreads. These classes are provided explicitly to enable communication between the real-time execution of NoHeapRealtimeThreads and regular Java threads.

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 handleAsyncEvent() methods of instances of AsyncEventHandler are scheduled and thus perform the required logic.~~ 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 AsyncEvent manages two things: 1) the unblocking of handlers when the event is fired, and 2) the set of handlers associated with the event. This set can be queried, have handlers added, or have handlers removed.~~

An instance of AsyncEventHandler can be thought of as something roughly similar to a thread. It is a Runnable object: when the event fires, ~~the handleAsyncEvent() methods of the associated handlers are scheduled.~~ 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 off 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 soft 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 thread needs to explicitly indicate its susceptibility to ATC. Since legacy code or library methods might have been written assuming no ATC, by default ATC should be turned off (more precisely, it should be deferred as long as control is in such code).
- Even if a thread allows ATC, some code sections need to be executed to completion and thus ATC is deferred in such sections. The ATC-deferred sections are synchronized methods and synchronized statements.
- Code that responds to an ATC does not return to the point in the thread 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 thread. This triggering may be direct (from a source thread) 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. In particular, it must be possible to base an ATC on a timer going off.
- Through ATC it must be possible to abort a 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 timeout 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 timeout 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 thread termination.
- ATC must be implemented without inducing an overhead for programs that do not use it.
- ~~If code with a timeout completes before the timeout's deadline, the timeout needs to be automatically stopped and its resources returned to the system.~~ If code with a timeout completes before the timer's expiration, the timer needs to be automatically stopped and its resources returned to the system.

Asynchronous Thread Termination

Although not a real-time issue, many event-driven computer systems that tightly interact with external real-world noncomputer systems (e.g., humans, machines, control processes, etc.) may require ~~significant~~ mode changes in their computational behavior as a result of significant changes in the non-computer real-world system. It is 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. ~~Consider the opposite case. A~~ Without this facility, a thread or set of threads would 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 has not yet been deprecated, its usage is 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 thread termination through a combination of the asynchronous event handling and the asynchronous transfer of control mechanisms. ~~If the significantly long or blocking methods of a thread are made interruptible the oracle can consist of a number of asynchronous event handlers that are bound to external happenings. When the happenings occur the handlers can invoke interrupt() on appropriate threads. Those threads will then clean up by having all of the interruptible methods transfer control to appropriate catch clauses as control enters those methods (either by invocation or by the return bytecode). This continues until the run() method of the thread returns.~~ To create such a set of threads consider the following steps:

- Make all of the application methods of the thread interruptible
- Create an oracle which monitors the external world by binding a number of asynchronous even handlers to happenings which occur at appropriate mode changes
- Have the handlers call interrupt() on each of the threads affected by the change
- After the handlers call interrupt() have them create a new set of 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 thread will complete normally.

This idiom provides a quick (if coded to be so) but orderly clean up and termination of the 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. ~~RawMemoryAccess defines methods that allow the programmer to construct an object that represents a range of physical addresses and then access the physical memory with byte, short, int, long, float, and double granularity.~~ 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&lttype>() and set&lttype>() 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&lttype>()` and `get&lttype>()` methods are implied. ~~The VTPhysicalMemory, LTPhysicalMemory and ImmortalPhysicalMemory classes allow programmers to create objects that represent a range of physical memory addresses and in which Java objects can be located.~~ 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. The base implementation will provide a `PhysicalMemoryManager` and a set of `PhysicalMemoryTypeFilter` classes that correctly identify memory classes that are standard for the (OS, JVM, and processor) platform. 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 addresses the high-order or low-order byte is based on the value of the BYTE_ORDER static boolean variable in class RealtimeSystem.~~ 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 lowlevel 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 errorprone (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.

CHAPTER 3

Conventions

Parameter Objects

A number of constructors in this specification take objects generically named `feasibility parameters` (classes named `<string>Parameters` where `<string>` identifies the kind of parameter). When a reference to a `Parameters` object is given as a parameter to a constructor the `Parameters` object becomes bound to the object being created. Changes to the values in the `Parameters` object affect the constructed object. For example, if a reference to a `SchedulingParameters` object, `sp`, is given to the constructor of a `RealtimeThread`, `rt`, then calls to `sp.setPriority()` will change the priority of `rt`. There is no restriction on the number of constructors to which a reference to a single `Parameters` object may be given. If a `Parameters` object is given to more than one constructor, then changes to the values in the `Parameters` object affect *all* of the associated schedulable objects. Note that this is a one-to-many relationship, *not* a many-to-many relationship, that is, a schedulable object (e.g., an instance of `RealtimeThread`) must have zero or one associated instance of each `Parameter` object type.

Caution: `<string>Parameter` objects 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.

Java Platform Dependencies

In some cases the classes and methods defined in this specification are dependent on the underlying Java platform.

1. The `Comparable` interface is available in Java™ 2 v1.2 1nd 1.3 and not in what was formally known as JDK's 1.0 and 1.1. Thus, we expect implementations of this specification which are based on JDK's 1.0 or 1.1 to include a `Comparable` interface.

2. The class `RawMemoryFloatAccess` is required if and only if the underlying Java Virtual Machine supports floating point data types.

Illegal Parameter Values

Except as noted explicitly in the descriptions of constructors, methods, and parameters an instance of IllegalArgumentException will be thrown if the value of the parameter or of a field of an instance of an object given as a parameter is as given in the following table:

Type	**Value**
Object	null
type[]	null
String	Null
int	less than zero
long	less than zero
float	less than zero
boolean	N/A
Class	null

Explicit exceptions to these semantics may also be global at the Chapter, Class, or Method level.

CHAPTER 4

Conformance, Compliance, and Portability of the RTSJ

Minimum Implementations of the RTSJ

The flexibility of the RTSJ indicates that implementations may provide different semantics for scheduling, synchronization, and garbage collection. This section defines what minimum semantics for these areas and other semantics and APIs required of all implementations of the RTSJ. In general, the RTSJ does not allow any subsetting of the APIs in the `javax.realtime` package (except those noted as optionally required); however, some of the classes are specific to certain well-known scheduling or synchronization algorithms and may have no underlying support in a minimum implementation of the RTSJ. The RTSJ provides these classes as standard parent classes for implementations supporting such algorithms.

The minimum scheduling semantics that must be supported in all implementations of the RTSJ are fixed-priority preemptive scheduling and at least 28 unique priority levels. By fixed-priority we mean that the system does not change the priority of any `RealtimeThread` or `NoHeapRealtimeThread` except, temporarily, for priority inversion avoidance. Note, however, that application code may change such priorities. What the RTSJ precludes by this statement is scheduling algorithms that change thread priorities according to policies for optimizing throughput (such as increasing the priority of threads that have been receiving few processor cycles because of higher priority threads (aging)). The 28 unique priority levels are required to be unique to preclude implementations from using fewer priority levels of underlying systems to implement the required 28 by simplistic algorithms (such as lumping four RTSJ priorities into seven buckets for an underlying system that only supports seven priority levels). It is sufficient for systems with fewer than 28 priority levels to use more sophisticated algorithms to implement the required 28 unique levels as long as `RealtimeThreads` and `NoHeapRealtimeThreads` behave as though there were at least 28 unique levels. (e.g. if there were 28 RealtimeThreads (t_1,...,t_{28}) with priorities (p_1,...,p_{28}), respectively, where the value of p_1 was the highest priority and the value of p_2 the next highest priority, etc., then for all executions of threads t_1 through t_{28} thread t_1 would *always* execute in preference to threads t_2, ..., t_{28} and thread t_2 would *always* execute in preference to threads t_3,..., t_{28}, etc.)

The minimum synchronization semantics that must be supported in all implementations of the RTSJ are detailed in the above section on synchronization and repeated here.

All implementations of the RTSJ 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.

All threads waiting to acquire a resource must be queued in priority order. This applies to the processor as well as to synchronized blocks. If threads with the same exact priority are possible under the active scheduling policy, threads with the same priority are queued in FIFO order. (Note that these requirements apply only to the required base scheduling policy and hence use the specific term “priority”). In particular:

- Threads waiting to enter synchronized blocks are granted access to the synchronized block in priority order.
- A blocked thread that becomes ready to run is given access to the processor in priority order.
- A thread whose execution eligibility is explicitly set by itself or another thread is given access to the processor in priority order.
- A thread that performs a `yield()` will be given access to the processor after waiting threads of the same priority.
- However, threads that are preempted in favor of a thread with higher priority 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.

The RTSJ does not require any particular garbage collection algorithm. All implementations of the RTSJ must, however, support the class `GarbageCollector` and implement all of its methods.

Optionally Required Components

The RTSJ does not, in general, support the concept of optional components of the specification. Optional components would further complicate the already difficult task of writing WORA (Write Once Run Anywhere) software components for real-time systems. However, understanding the difficulty of providing implementations of mechanisms for which there is no underlying support, the RTSJ does provide for a few exceptions. Any components that are considered optional will be listed as such in the class definitions.

The most notable optional component of the specification is the `POSIXSignalHandler`. A conformant implementation must support POSIX signals if and only if the underlying system supports them. Also, the class `RawMemoryFloatAccess` is required to be implemented if and only if the JVM itself supports floating point types.

Documentation Requirements

In order to properly engineer a real-time system, an understanding of the cost associated with any arbitrary code segment is required. This is especially important for operations that are performed by the runtime system, largely hidden from the programmer. (An example of this is the maximum expected latency before the garbage collector can be interrupted.)

The RTSJ does not require specific performance or latency numbers to be matched. Rather, to be conformant to this specification, an implementation must provide documentation regarding the expected behavior of particular mechanisms. The mechanisms requiring such documentation, and the specific data to be provided, will be detailed in the class and method definitions.

CHAPTER 5

Threads

This section contains classes that:

- Provide for the creation of threads that have more precise scheduling semantics than `java.lang.Thread`.
- Allow the use of areas of memory other than the heap for the allocation of objects.
- Allow the definition of methods that can be asynchronously interrupted.
- Provide the scheduling semantics for handling asynchronous events.

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

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.

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 default scheduling policy must manage the execution of instances which implement the interface `Schedulable`.
2. Any scheduling policy present in an implementation must be available to instances of objects which implement the interface `Schedulable`.

3. The function of allocating objects in memory in areas defined by instances of `ScopedMemory` or its subclasses shall be available only to logic within instances of `RealtimeThread`, `NoHeapRealtimeThread`, `AsyncEventHandler`, and `BoundAsyncEventHandler`.
4. The invocation of methods that throw `AsynchronouslyInterruptedException` has the indicated effect only when the invocation occurs in the context of instances of `RealtimeThread`, `NoHeapRealtimeThread`, `AsyncEventHandler`, and `BoundAsyncEventHandler`.
5. ~~Instances of the NoHeapRealtimeThread class have an implicit execution eligibility logically higher than any garbage collector.~~ Instances of the NoHeapRealtimeThread class may have an execution eligibility higher than any garbage collector. If the garbage collector executes in the context of a thread with a higher execution eligibility then the instance will block because the RTSJ honors execution eligibility first. However, unlike instances of java.lang.Thread or RealtimeThread, the instance of NoHeapRealtimeThread will not be blocked by the garbage collector executing in the context of a thread with a lower execution eligibility.
6. Instances of the `RealtimeThread` class may have an execution eligibility logically lower than the garbage collector.
7. Changing values in `SchedulingParameters, ProcessingParameters, ReleaseParameters, ProcessingGroupParameters`, or use of `Thread.setPriority()` must not affect the correctness of any implemented priority inversion avoidance algorithm.
8. Instances of objects which implement the interface `Schedulable` will inherit the scope stack (see the Memory Chapter) of the thread invoking the constructor. If the thread invoking the constructor does not have a scope stack then the scope stack of the new object will have one entry which will be the current allocation context of the thread invoking the constructor.
9. Instances of objects which implement the interface `Schedulable` will have an initial entry in their scope stack. This entry will be either: the memory area given as a parameter to the constructor, or, if no memory area is given, the allocation context of the thread invoking the constructor.
10. The default parameter values for an object implmenting the interface `Schedulable` will be the parameter values of the thread invoking the constructor. If the thread invoking the constructor does not have parameter values then the default values are those values associated with the instance of `Scheduler` which will manage the object.
11. Instance of objects implementing the interface `Schedulable` can be placed in memory represented by instances of `ImmortalMemory, HeapMemory, Scoped-`

PhysicalMemory, or PhysicalImmortal.

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 merely extending `java.lang.Thread`. The `RealtimeParameters` 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. 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.~~ 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
            Schedulable47
```

All Implemented Interfaces: `java.lang.Runnable`, `Schedulable`$_{47}$

Direct Known Subclasses: `NoHeapRealtimeThread`$_{38}$

Description

Class `RealtimeThread` extends `java.lang.Thread` and includes classes and methods to get and set parameter objects, manage the execution of those threads with a `ReleaseParameters`$_{66}$ type of `PeriodicParameters`$_{70}$, and manage waiting.

A `RealtimeThread`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 such execution would cause access violations.

Parameters for constructors may be `null`. In such cases the default value will be the default value set for the particular type by the associated instance of `Scheduler`54 .

5.1.1 Constructors

RealtimeThread

`public RealtimeThread()`

Create a real-time thread. All parameter values are null.

RealtimeThread

`public RealtimeThread(SchedulingParameters63 scheduling)`

Create a real-time thread with the given `SchedulingParameters`63 .

Parameters:

`scheduling` - The `SchedulingParameters`63 associated with `this` (and possibly other instances of `Schedulable`47).

RealtimeThread

```
public RealtimeThread(SchedulingParameters63 scheduling,
      ReleaseParameters66 release)
```

Create a real-time thread with the given `SchedulingParameters`63 and `ReleaseParameters`66 .

Parameters:

`scheduling` - The `SchedulingParameters`63 associated with `this` (and possibly other instances of `Schedulable`47).

`release` - The `ReleaseParameters`66 associated with `this` (and possibly other instances of `Schedulable`47).

RealtimeThread

```
public RealtimeThread(SchedulingParameters63 scheduling,
      ReleaseParameters66 release, MemoryParameters150 memory,
      MemoryArea90 area, ProcessingGroupParameters81 group,
      java.lang.Runnable logic)
```

Create a real-time thread with the given characteristics and a `java.lang.Runnable` .

Parameters:

`scheduling` - The `SchedulingParameters`$_{63}$ associated with `this` (An possibly other instances of `Schedulable`$_{47}$).

`release` - The `ReleaseParameters`$_{66}$ associated with `this` (and possibly other instances of `Schedulable`$_{47}$).

`memory` - The `MemoryParameters`$_{150}$ associated with `this` (and possibly other instances of `Schedulable`$_{47}$).

`area` - The `MemoryArea`$_{90}$ associated with `this`.

`group` - The `ProcessingGroupParameters`$_{81}$ associated with this (and possibly other instances of `Schedulable`$_{47}$).

5.1.2 Methods

addIfFeasible

```
public boolean addIfFeasible()
```

Add the scheduling and release charactics of `this` to the set of such characteristics already being considered, if the addition would result in the new, larger set being feasible.

Specified By: `addIfFeasible`$_{47}$ in interface `Schedulable`$_{47}$

Returns: True, if the addition would result in the set of considered characteristics being feasible. False, if the addition would result in the set of considered characteristics being infeasible or there is no assigned instance of `Scheduler`$_{54}$.

addToFeasibility

```
public boolean addToFeasibility()
```

Inform the scheduler and cooperating facilities that scheduling and release characteristics of this instance of `Schedulable`$_{47}$ should be considered in feasibility analysis until further notified.

Specified By: `addToFeasibility`$_{48}$ in interface `Schedulable`$_{47}$

Returns: True, if the addition was successful. False, if not.

currentRealtimeThread

```
public static RealtimeThread25 currentRealtimeThread()
    throws ClassCastException
```

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

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

Throws:

`java.lang.ClassCastException` - If the current thread is not a `RealtimeThread`.

deschedulePeriodic

`public void` **`deschedulePeriodic`**`()`

Stop unblocking `waitForNextPeriod()`38 for `this` if the type of the associated instance of `ReleaseParameters`66 is `PeriodicParameters`70 If `this` does not have a type of `PeriodicParameters`70 , nothing happens.

getCurrentMemoryArea

`public static MemoryArea`90 **`getCurrentMemoryArea`**`()`

Gets the current memory area of `this`.

Returns: A reference to the current `MemoryArea`90 object.

getInitialMemoryAreaIndex

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

Memory area stacks include inherited stacks from parent threads. The inital memory area for the current `RealtimeThread` is the memory area given as a parameter to the constructor. This method returns the position in the memory area stack of that initial memory area.

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

getMemoryArea

`public MemoryArea`90 **`getMemoryArea`**`()`

Gets a reference to the current `MemoryArea`90 .

Returns: A reference to the current memory area in which allocations occur.

getMemoryAreaStackDepth

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

Gets the size of the stack of `MemoryArea`90 instances to which this `RealtimeThread` has access.

Returns: The size of the stack of `MemoryArea`$_{90}$ instances.

getMemoryParameters

`public MemoryParameters`$_{150}$ **`getMemoryParameters`**`()`

Gets a reference to the `MemoryParameters`$_{150}$ object.

Specified By: `getMemoryParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: A reference to the current `MemoryParameters`$_{150}$ object.

getOuterMemoryArea

`public static MemoryArea`$_{90}$ **`getOuterMemoryArea`**`(int index)`

Gets the instance of `MemoryArea`$_{90}$ 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.

Parameters:

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

Returns: The instance of `MemoryArea`$_{90}$ at index or `null` if the given value is does not correspond to a position in the stack.

getProcessingGroupParameters

`public ProcessingGroupParameters`$_{81}$ **`getProcessingGroupParameters`**`()`

Gets a reference to the `ProcessingGroupParameters`$_{81}$ object.

Specified By: `getProcessingGroupParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: A reference to the current `ProcessingGroupParameters`$_{81}$ object.

getReleaseParameters

`public ReleaseParameters`$_{66}$ **`getReleaseParameters`**`()`

Gets a reference to the `ReleaseParameters`$_{66}$ object.

Specified By: `getReleaseParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: A reference to the current `ReleaseParameters`$_{66}$ object.

getScheduler

`public Scheduler`$_{54}$ **`getScheduler`**`()`

Gets a reference to the `Scheduler`54 object.

Specified By: `getScheduler`48 in interface `Schedulable`47

Returns: A reference to the current `Scheduler`54 object.

getSchedulingParameters

`public SchedulingParameters`63 **`getSchedulingParameters`**`()`

Gets a reference to the `SchedulingParameters`63 object.

Specified By: `getSchedulingParameters`48 in interface `Schedulable`47

Returns: A reference to the current `SchedulingParameters`63 object.

interrupt

`public void` **`interrupt`**`()`

Sets the state of the generic `AsynchronouslyInterruptedException`226 to pending.

Overrides: `interrupt` in class `Thread`

removeFromFeasibility

`public boolean` **`removeFromFeasibility`**`()`

Inform the scheduler and cooperating facilities that the scheduling and release characteristics of this instance of `Schedulable`47 should *not* be considered in feasibility analyses until further notified.

Specified By: `removeFromFeasibility`49 in interface `Schedulable`47

Returns: True, if the removal was successful. False, if the removal was unsuccessful.

schedulePeriodic

`public void` **`schedulePeriodic`**`()`

Begin unblocking `waitForNextPeriod()`38 for a periodic thread. Typically used when a periodic schedulable object is in an overrun condition. The scheduler should recompute the schedule and perform admission control. If this does not have a type of `PeriodicParameters`70 as it `ReleaseParameters`66 nothing happens.

setIfFeasible

`public boolean setIfFeasible(ReleaseParameters66 release, MemoryParameters150 memory)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable47` or an instance of `Schedulable47`. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable47`. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable47` as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible49` in interface `Schedulable47`

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

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.

setIfFeasible

`public boolean setIfFeasible(ReleaseParameters66 release, MemoryParameters150 memory, ProcessingGroupParameters81 group)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable47` or an instance of `Schedulable47`. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable47`. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable47` as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible49` in interface `Schedulable47`

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

`group` - The proposed processing group parameters.

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.

setIfFeasible

`public boolean` **`setIfFeasible`**`(ReleaseParameters`$_{66}$ `release, ProcessingGroupParameters`$_{81}$ `group)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible`$_{50}$ in interface `Schedulable`$_{47}$

Parameters:

`release` - The proposed release parameters.

`group` - The proposed processing group parameters.

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.

setMemoryParameters

`public void` **`setMemoryParameters`**`(MemoryParameters`$_{150}$ `parameters) throws IllegalThreadStateException`

Sets the memory parameters associated with this instance of `Schedulable`$_{47}$.

Specified By: `setMemoryParameters`$_{50}$ in interface `Schedulable`$_{47}$

Parameters:

`memory` - A `MemoryParameters`$_{150}$ object which will become the memory parameters associated with `this` after the method call.

Throws:

`java.lang.IllegalThreadStateException`

setMemoryParametersIfFeasible

```
public boolean setMemoryParametersIfFeasible(MemoryParameters150
        memory)
```

The method first performs a feasibility analysis using the given memory parameters as replacements for the memory parameters of `this` If the resulting system is feasible the method replaces the current memory parameters of `this` with the new memory parameters.

Specified By: `setMemoryParametersIfFeasible`51 in interface `Schedulable`47

Parameters:

`memory` - The proposed memory parameters. If null, nothing happens.

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.

setProcessingGroupParameters

```
public void
        setProcessingGroupParameters(ProcessingGroupParameters81
        parameters)
```

Sets the reference to the `ProcessingGroupParameters`81 object.

Specified By: `setProcessingGroupParameters`51 in interface `Schedulable`47

Parameters:

`parameters` - A reference to the `ProcessingGroupParameters`81 object.

setProcessingGroupParametersIfFeasible

```
public boolean
        setProcessingGroupParametersIfFeasible(ProcessingGroupParam
        eters81 group)
```

Sets the `ProcessingGroupParameters`81 of `this` only if the resulting set of scheduling and release characteristics is feasible.

Specified By: `setProcessingGroupParametersIfFeasible`51 in interface `Schedulable`47

Parameters:

`group` - The `ProcessingGroupParameters`81 object. If null, nothing happens.

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.

setReleaseParameters

```
public void setReleaseParameters(ReleaseParameters66 release)
    throws IllegalThreadStateException
```

Sets the release parameters associated with this instance of `Schedulable`47.

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

Specified By: `setReleaseParameters`52 in interface `Schedulable`47

Parameters:

`release` - A `ReleaseParameters`66 object which will become the release parameters associated with this after the method call.

Throws:

`java.lang.IllegalThreadStateException` - Thrown is the state of this instance of `Schedulable`47 is not appropriate to changing the release parameters.

setReleaseParametersIfFeasible

```
public boolean setReleaseParametersIfFeasible(ReleaseParameters66
    release)
```

Set the `ReleaseParameters`66 for this schedulable object only if the resulting set of scheduling and release characteristics is feasible.

Specified By: `setReleaseParametersIfFeasible`52 in interface `Schedulable`47

Parameters:

`release` - The `ReleaseParameters`66 object. If null, nothing happens.

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.

setScheduler

```
public void setScheduler(Scheduler54 scheduler)
        throws IllegalThreadStateException
```

Sets the reference to the `Scheduler`54 object.

Specified By: `setScheduler`52 in interface `Schedulable`47

Parameters:

`scheduler` - A reference to the `Scheduler`54 object.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setScheduler

```
public void setScheduler(Scheduler54 scheduler,
        SchedulingParameters63 scheduling,
        ReleaseParameters66 release, MemoryParameters150 memory,
        ProcessingGroupParameters81 group)
        throws IllegalThreadStateException
```

Sets the scheduler and associated parameter objects.

Specified By: `setScheduler`53 in interface `Schedulable`47

Parameters:

`scheduler` - A reference to the scheduler that will manage the execution of this thread. If null, no change to current value of this parameter is made.

`scheduling` - A reference to the `SchedulingParameters`63 which will be associated with `this`. If null, no change to current value of this parameter is made.

`release` - A reference to the `ReleaseParameters`66 which will be associated with `this`. If null, no change to current value of this parameter is made.

`memory` - A reference to the `MemoryParameters`150 which will be associated with `this`. If null, no change to current value of this parameter is made.

`group` - A reference to the `ProcessingGroupParameters`81 which will be associated with `this`. If null, no change to current value of this parameter is made.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setSchedulingParameters

```
public void setSchedulingParameters(SchedulingParameters63
    scheduling)
    throws IllegalThreadStateException
```

Sets the reference to the `SchedulingParameters`63 object.

Specified By: `setSchedulingParameters`53 in interface `Schedulable`47

Parameters:

`scheduling` - A reference to the `SchedulingParameters`63 object.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setSchedulingParametersIfFeasible

```
public boolean
    setSchedulingParametersIfFeasible(SchedulingParameters63
    scheduling)
```

The method first performs a feasibility analysis using the given scheduling parameters as replacements for the scheduling parameters of `this` If the resulting system is feasible the method replaces the current scheduling parameters of `this` with the new scheduling parameters.

Specified By: `setSchedulingParametersIfFeasible`54 in interface `Schedulable`47

Parameters:

`scheduling` - The proposed scheduling parameters. If null, nothing happens.

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.

sleep

```
public static void sleep(Clock192 clock,
    HighResolutionTime172 time)
    throws InterruptedException
```

An accurate timer with nanosecond granularity. The actual resolution available for the clock must be queried from somewhere else. The time base is the given `Clock`192 . The sleep time may be relative or absolute. If relative, then the calling thread is blocked for the amount of time given by the parameter. If absolute, then the calling thread is blocked until the indicated point in time. If the given absolute time is before the current time, the call to sleep returns immediately.

Parameters:

`clock` - The instance of `Clock`192 used as the base.

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

Throws:

`java.lang.InterruptedException` - If interrupted.

sleep

```
public static void sleep(HighResolutionTime172 time)
    throws InterruptedException
```

An accurate timer with nanosecond granularity. The actual resolution available for the clock must be queried from somewhere else. The time base is the default `Clock`192 . The sleep time may be relative or absolute. If relative, then the calling thread is blocked for the amount of time given by the parameter. If absolute, then the calling thread is blocked until the indicated point in time. If the given absolute time is before the current time, the call to sleep returns immediately.

Parameters:

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

Throws:

`java.lang.InterruptedException` - If interrupted.

start

```
public void start()
```

Starts the thread.

Overrides: `start` in class `Thread`

waitForNextPeriod

```
public boolean waitForNextPeriod()
      throws IllegalThreadStateException
```

Used by threads that have a reference to a `ReleaseParameters`66 type of `PeriodicParameters`70 to block until the start of each period. Periods start at either the start time in `PeriodicParameters`70 or when `this.start()` is called. This method will block until the start of the next period unless the thread is in either an overrun or deadline miss condition. If both overrun and miss handlers are null and the thread has overrun its cost or missed a deadline `waitForNextPeriod()`38 will immediately return false once per overrun or deadline miss. It will then again block until the start of the next period (unless, of course, the thread has overrun or missed again). If either the overrun or deadline miss handlers are not null and the thread is in either an overrun or deadline miss condition `waitForNextPeriod()`38 will block until the handler corrects the situation (possibly by calling `schedulePeriodic()`30).

Returns: True when the thread is not in an overrun or deadline miss condition and unblocks at the start of the next period.

Throws:

`java.lang.IllegalThreadStateException` - If `this` does not have a reference to a `ReleaseParameters`66 type of `PeriodicParameters`70 .

5.2 NoHeapRealtimeThread

Declaration

`public class` **`NoHeapRealtimeThread`** `extends RealtimeThread`25

All Implemented Interfaces: `java.lang.Runnable`, `Schedulable`47

Description

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`25 . 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 nor it is even allowed to manipulate the references to objects in the heap. For example, if a and b are objects in immortal memory, b.p is reference to an object on the heap, and a.p is type compatible with b.p, then a `NoHeapRealtimeThread` is *not* allowed to execute anyting like the following:

```
a.p = b.p; b.p = null;
```

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 ScopedMemory 98 or ImmortalMemory 96 .

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.

Parameters for constructors may be null. In such cases the default value will be the default value set for the particular type by the associated instance of Scheduler 54 .

5.2.1 Constructors

NoHeapRealtimeThread

```
public NoHeapRealtimeThread(SchedulingParameters63 scheduling,
       MemoryArea90 area)
       throws IllegalArgumentException
```

Create a NoHeapRealtimeThread.

Parameters:

scheduling - A SchedulingParameters 63 object that will be associated with this. A null value means this will not have an associated SchedulingParameters 63 object.

area - A MemoryArea 90 object. Must be a ScopedMemory 98 or ImmortalMemory 96 type. A null value causes an IllegalArgumentException to be thrown.

Throws:

java.lang.IllegalArgumentException - If the memory area parameter is null.

NoHeapRealtimeThread

```
public NoHeapRealtimeThread(SchedulingParameters63 scheduling,
       ReleaseParameters66 release, MemoryArea90 area)
       throws IllegalArgumentException
```

Create a NoHeapRealtimeThread.

Parameters:

`scheduling` - A `SchedulingParameters`*63* object that will be associated with this. A null value means this will not have an associated `SchedulingParameters`*63* object.

`release` - A `ReleaseParameters`*66* object that will be associated with this. A null value means this will not have an associated `ReleaseParameters`*66* object.

`area` - A `MemoryArea`*90* object. Must be a `ScopedMemory`*98* or `ImmortalMemory`*96* type. A null value causes an `IllegalArgumentException` to be thrown.

Throws:

`java.lang.IllegalArgumentException` - If the memory area parameter is null.

NoHeapRealtimeThread

```
public NoHeapRealtimeThread(SchedulingParameters63 scheduling,
      ReleaseParameters66 release, MemoryParameters150 memory,
      MemoryArea90 area, ProcessingGroupParameters81 group,
      java.lang.Runnable logic)
      throws IllegalArgumentException
```

Create a `NoHeapRealtimeThread`.

Parameters:

`scheduling` - A `SchedulingParameters`*63* object that will be associated with this. A null value means this will not have an associated `SchedulingParameters`*63* object.

`release` - A `ReleaseParameters`*66* object that will be associated with this. A null value means this will not have an associated `ReleaseParameters`*66* object.

`memory` - A `MemoryParameters`*150* object that will be associated with this. A null value means this will not have a `MemoryParameters`*150* object.

`area` - A `MemoryArea`*90* object. Must be a `ScopedMemory`*98* or `ImmortalMemory`*96* type. A null value causes an `IllegalArgumentException` to be thrown.

`group` - A `ProcessingGroupParameters`*81* object that will be associated with this. A null value means this will not have an associated `ProcessingGroupParameters`*81* object.

`logic` - A Runnable whose `run()` method will be executed for this.

Throws:

`java.lang.IllegalArgumentException` - If the memory area parameter is null.

5.2.2 Methods

start

`public void **start**()`

Checks if the `NoHeapRealtimeThread` is startable and starts it if it is. Checks that the parameters associated with this NHRT object are not allocated in heap. Also checks if `this` object is allocated in heap. If any of them are allocated, `start()` throws a `MemoryAccessError`247

Overrides: `start`37 in class `RealtimeThread`25

Throws:

`MemoryAccessError`247 - If any of the parameters or `this` is allocated on heap.

CHAPTER 6

Scheduling

This section contains classes that:

- Allow the definition of schedulable objects.
- Manage the assignment of execution eligibility to schedulable objects.
- Perform feasibility analysis for sets of schedulable objects.
- Control the admission of new schedulable objects.
- 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.
- Define temporal containers used to enforce correct temporal behavior of multiple schedulable objects.

The scheduler required by this specification is fixed-priority preemptive with 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`, and `AsyncEventHandler`. Each of these is assigned processor resources according to their release characteristics, execution eligibility, and processing group values. Any subclass of these objects or any class implementing the `Schedulable` interface are 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 in the `SchedulingParameters` object used in the constructor of the schedulable object. For implementations providing only the base scheduling policy, the previous statement holds for the specific type `PriorityParameters` (a subclass of

SchedulingParameters), Implementations providing additional scheduling policies or execution eligibility assignment policies which require an application visible field to contain execution eligibility then SchedulingParamters must be subclassed and the previous statement then holds for the specific subclass type. If, however, additionally provided scheduling policies or execution eligibility assignment policies do not require application visibility of execution eligibility or it appears in another parameter object (e.g., the earliest deadline first scheduling uses deadline as the execution eligibility metric and would thus be visible in ReleaseParameters), then SchedulingParameters need not be subclassed.

An instance of the ReleaseParameters class or its subclasses, PeriodicParameters, AperiodicParameters, and SporadicParameters, contains values that define a particular release discipline. A schedulable object is considered to have the release characteristics of a single associated instance of the ReleaseParameters class. In all cases the Scheduler uses these values to perform its feasibility analysis over the set of schedulable objects and admission control for the schedulable object. Additionally, for those schedulable objects whose associated instance of ReleaseParameters is an instance of PeriodicParameters, the scheduler manages the behavior of the object's waitForNextPeriod() method and monitors overrun and deadline-miss conditions. In the case of overrun or deadline-miss the scheduler changed the behavior of the waitForNextPeriod()and schedules the appropriate handler.

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 schedulable objects. The combined processor demand of all of the schedulable objects associated with an instance of the ProcessingParameters 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 list establishes the semantics and requirements that are applicable across the classes of this section and 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.

1. The base scheduler must support at least 28 unique values in the priorityLevel field of an instance of PriorityParameters.
2. Higher values in the priorityLevel field of an instance of Priority-

Parameters have a higher execution eligibility.

3. In (1) unique means that if two schedulable objects have different values in the `priorityLevel` field in their respective instances of `PriorityParameters`, the schedulable object with the higher value will always execute in preference to the schedulable object with the lower value when both are ready to execute.
4. An implementation must make available some native priorities which are lower than the 28 required real-time priorities. These are to be used for regular Java threads (i.e., instances of threads which are not instances of `RealtimeThread`, `NoHeapRealtimeThread`, or `AsyncEventHandler` classes or subclasses). The ten traditional Java thread priorities may have an arbitrary mapping into the native priorities. These ten traditional Java thead priorities and the required minimum 28 unique real-time thread priorities shall be from the same space. Assignment of any of these (minimum) 38 priorities to real-time threads or traditional Java threads is legal. It is the responsibility of application logic to make rational priority assignments.
5. The dispatching mechanism must allow the preemption of the execution of schedulable objects at a point not governed by the preempted object.
6. For schedulable objects managed by the base scheduler no part of the system may change the execution eligibility for any reason other than implementation of a priority inversion algorithm. This does not preclude additional schedulers from changing the execution eligibility of schedulable objects—which they manage—according to the scheduling algorithm.
7. Threads that are preempted in favor of a higher priority thread may be placed in the appropriate queue at any position as determined by a particular implementation. The implementation is required to provide documentation stating exactly the algorithm used for such placement.
8. If an implementation provides any schedulers other than the base scheduler it shall provide documentation explicitly stating the semantics expressed by 8 through 11 in language and constructs appropriate to the provided scheduling algorithms.
9. All instances of `RelativeTime` used in instances of `ProcessingParameters`, `SchedulingParameters`, and `ReleaseParameters` are measured from the time at which the associated thread (or first such thread) is started.
10. `PriorityScheduler.getNormPriority()` shall be set to `((PriorityScheduler.getMaxPriority() - PriorityScheduler.getMinPriority())/3) + PriorityScheduler.getMinPriority()`.
11. If instances of `RealtimeThread or NoHeapRealtimeThread` are constructed without a reference to a SchedulingParameters object a `SchedulingParamters`

object is created and assigned the values of the current thread. This does not imply that other schedulers should follow this rule. Other schedulers are free to define the default scheduling parameters in the absence of a given `SchedulingParameters` object.

12. The policy and semantics embodied in 1 through 11 above and by the descriptions of the refered to classes, methods, and their interactions must be available in all implementations of this specification.
13. This specification does not require any particular feasibility algorithm be implemented in the `Scheduler` object. Those implementations that choose to not implement a feasibility algorithm shall return success whenever the feasibility algorithm is executed.
14. Implementations that provide a scheduler with a feasibility algorithm are required to clearly document the behavior of that algorithm.
15. For instances of `AsyncEventHandler` with a release parameters object of type `SporadicParameters` implementations are required to maintain a list of times at which instances of `AsyncEvent` occurred. The i^{th} time may be removed from the queue after the i^{th} execution of the `handleAsyncEvent` method.
16. If the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters objects of type `SporadicParameters` attached and the execution of `AsyncEvent.fire()` introduces the requirement to throw at least one type of exception, then all instance of `AsyncEventHandler` not affected by the exception are handled normally.
17. If the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters objects 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 `MITViolationException` has precedence over `ResourceLimitExceeded`.

The following hold for the `PriorityScheduler`:

1. A blocked thread that becomes ready to run is added to the tail of any runnable queue for that priority.
2. For a thread whose effective priority is changed as a result of explicitly setting `priorityLevel` this thread or another thread is added to the tail of the runnable queue for the new `priorityLevel`.
3. A thread that performs a `yield()` goes to the tail of the runnable queue for its `priorityLevel`.

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. By requirement ~~16~~ 6, the specification accommodates existing practice, which is a stated goal of the effort.

The semantics of the classes, constructors, methods, and fields within allow for the natural extension of the scheduling policy by implementations that provide different scheduler objects.

Some research shows that, given a set of reasonable common assumptions, 32 unique priority levels are a reasonable choice for close-to-optimal scheduling efficiency when using the rate-monotonic priority assignment algorithm (256 priority levels better provide better efficiency). This specification requires at least 28 unique 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.

6.1 Schedulable

Declaration

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

All Superinterfaces: `java.lang.Runnable`

All Known Implementing Classes: `AsyncEventHandler`$_{210}$, `RealtimeThread`$_{25}$

Description

Handlers and other objects can be run by a `Scheduler`$_{54}$ if they provide a `run()` method and the methods defined below. The `Scheduler`$_{54}$ uses this information to create a suitable context to execute the `run()` method.

6.1.1 Methods

addIfFeasible

`public boolean` **`addIfFeasible`**`()`

Add the scheduling and release charactics of `this` to the set of such characteristics already being considered, if the addition would result in the new, larger set being feasible.

Returns: True, if the addition would result in the set of considered characteristics being feasible. False, if the addition would result in the set of considered characteristics being infeasible or there is no assigned instance of `Scheduler`$_{54}$.

addToFeasibility

`public boolean` **`addToFeasibility`**`()`

Inform the scheduler and cooperating facilities that scheduling and release characteristics of this instance of `Schedulable`$_{47}$ should be considered in feasibility analysis until further notified.

Returns: True, if the addition was successful. False, if not.

getMemoryParameters

`public MemoryParameters`$_{150}$ **`getMemoryParameters`**`()`

Gets a reference to the `MemoryParameters`$_{150}$ object.

Returns: A reference to the current `MemoryParameters`$_{150}$ object.

getProcessingGroupParameters

`public ProcessingGroupParameters`$_{81}$ **`getProcessingGroupParameters`**`()`

Gets a reference to the `ProcessingGroupParameters`$_{81}$ object.

Returns: A reference to the current `ProcessingGroupParameters`$_{81}$ object.

getReleaseParameters

`public ReleaseParameters`$_{66}$ **`getReleaseParameters`**`()`

Gets a reference to the `ReleaseParameters`$_{66}$ object.

Returns: A reference to the current `ReleaseParameters`$_{66}$ object.

getScheduler

`public Scheduler`$_{54}$ **`getScheduler`**`()`

Gets a reference to the `Scheduler`$_{54}$ object.

Returns: A reference to the current `Scheduler`$_{54}$ object.

getSchedulingParameters

`public SchedulingParameters`$_{63}$ **`getSchedulingParameters`**`()`

Gets a reference to the `SchedulingParameters`63 object.

Returns: A reference to the current `SchedulingParameters`63 object.

removeFromFeasibility

`public boolean` **`removeFromFeasibility`**`()`

Inform the scheduler and cooperating facilities that scheduling and release characteristics of this instance of `Schedulable`47 should *not* be considered in feasibility analysis until further notified.

Returns: True, if the removal was successful. False, if not.

setIfFeasible

`public boolean` **`setIfFeasible`**`(ReleaseParameters`66 `release, MemoryParameters`150 `memory)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

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.

setIfFeasible

`public boolean` **`setIfFeasible`**`(ReleaseParameters`66 `release, MemoryParameters`150 `memory, ProcessingGroupParameters`81 `group)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

`group` - The proposed processing group parameters.

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.

setIfFeasible

```
public boolean setIfFeasible(ReleaseParameters66 release,
    ProcessingGroupParameters81 group)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`release` - The proposed release parameters.

`group` - The proposed processing group parameters.

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.

setMemoryParameters

```
public void setMemoryParameters(MemoryParameters150 memory)
```

Sets the memory parameters associated with this instance of `Schedulable`47.

Parameters:

`memory` - A `MemoryParameters`$_{150}$ object which will become the memory parameters associated with `this` after the method call.

setMemoryParametersIfFeasible

`public boolean` **`setMemoryParametersIfFeasible`**`(MemoryParameters`$_{150}$ `memory)`

The method first performs a feasibility analysis using the given memory parameters as replacements for the memory parameters of `this` If the resulting system is feasible the method replaces the current memory parameters of `this` with the new memory parameters.

Parameters:

`memory` - The proposed memory parameters.

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.

setProcessingGroupParameters

`public void` **`setProcessingGroupParameters`**`(ProcessingGroupParameters`$_{81}$ `group)`

Sets the `ProcessingGroupParameters`$_{81}$ of `this` only if the resulting set of scheduling and release characteristics is feasible.

Parameters:

`group` - The `ProcessingGroupParameters`$_{81}$ object. If null, nothing happens.

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.

setProcessingGroupParametersIfFeasible

`public boolean` **`setProcessingGroupParametersIfFeasible`**`(ProcessingGroupParameters`$_{81}$ `groupParameters)`

Sets the `ProcessingGroupParameters`$_{81}$ of `this` only if the resulting set of scheduling and release characteristics is feasible.

Parameters:

`group` - The `ProcessingGroupParameters`$_{81}$ object. If null, nothing happens.

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.

setReleaseParameters

```
public void setReleaseParameters(ReleaseParameters66 release)
```

Sets the release parameters associated with this instance of `Schedulable`$_{47}$.

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

Parameters:

`release` - A `ReleaseParameters`$_{66}$ object which will become the release parameters associated with this after the method call.

setReleaseParametersIfFeasible

```
public boolean setReleaseParametersIfFeasible(ReleaseParameters66
    release)
```

Set the `ReleaseParameters`$_{66}$ for this schedulable object only if the resulting set of scheduling and release characteristics is feasible.

Parameters:

`release` - The `ReleaseParameters`$_{66}$ object. If null, nothing happens.

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.

setScheduler

```
public void setScheduler(Scheduler54 scheduler)
    throws IllegalThreadStateException
```

Sets the reference to the `Scheduler`$_{54}$ object.

Parameters:

`scheduler` - A reference to the `Scheduler`$_{54}$ object.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setScheduler

```
public void setScheduler(Scheduler54 scheduler,
     SchedulingParameters63 scheduling,
     ReleaseParameters66 release,
     MemoryParameters150 memoryParameters,
     ProcessingGroupParameters81 group)
     throws IllegalThreadStateException
```

Sets the scheduler and associated parameter objects.

Parameters:

`scheduler` - A reference to the scheduler that will manage the execution of this thread. If null, no change to current value of this parameter is made.

`scheduling` - A reference to the `SchedulingParameters`63 which will be associated with `this`. If null, no change to current value of this parameter is made.

`release` - A reference to the `ReleaseParameters`66 which will be associated with `this`. If null, no change to current value of this parameter is made.

`memory` - A reference to the `MemoryParameters`150 which will be associated with `this`. If null, no change to current value of this parameter is made.

`group` - A reference to the `ProcessingGroupParameters`81 which will be associated with `this`. If null, no change to current value of this parameter is made.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setSchedulingParameters

```
public void setSchedulingParameters(SchedulingParameters63
     scheduling)
```

Sets the reference to the `SchedulingParameters`63 object.

Parameters:

`scheduling` - A reference to the `SchedulingParameters`63 object.

Throws:

`java.lang.IllegalThreadStateException` - Thrown when: `((Thread.isAlive() && Not Blocked) == true)`. (Where blocked means waiting in `Thread.wait()`, `Thread.join()`, or `Thread.sleep()`)

setSchedulingParametersIfFeasible

```
public boolean
    setSchedulingParametersIfFeasible(SchedulingParameters63
    scheduling)
```

The method first performs a feasibility analysis using the given scheduling parameters as replacements for the scheduling parameters of `this` If the resulting system is feasible the method replaces the current scheduling parameters of `this` with the new scheduling parameters.

Parameters:

`scheduling` - The proposed scheduling parameters.

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.

6.2 Scheduler

Declaration

`public abstract class **Scheduler**`

Direct Known Subclasses: `PriorityScheduler`58

Description

An instance of `Scheduler` manages the execution of schedulable objects and may implement a feasibility algorithm. The feasibility algorithm may determine if the known set of schedulable objects, given their particular execution ordering (or priority assignment), is a feasible schedule. 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()`55 (e.g., `EDFScheduler`).

6.2.1 Constructors

Scheduler

`protected` **`Scheduler`**`()`

Create an instance of `Scheduler`.

6.2.2 Methods

addToFeasibility

`protected abstract boolean` **`addToFeasibility`**`(Schedulable`$_{47}$ `schedulable)`

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`$_{63}$, `ReleaseParameters`$_{66}$, `MemoryParameters`$_{150}$, and `ProcessingGroupParameters`$_{81}$) of the given instance of `Schedulable`$_{47}$ will be considered in the feasibility analysis of the associated `Scheduler`$_{54}$ until further notice. Whether the resulting system is feasible or not, the addition is completed.

Parameters:

`schedulable` - A reference to the given instance of `Schedulable`$_{47}$

Returns: True, if the addition was successful. False, if not.

fireSchedulable

`public abstract void` **`fireSchedulable`**`(Schedulable`$_{47}$ `schedulable)`

Trigger the execution of a schedulable object (like an `AsyncEventHandler`$_{210}$).

Parameters:

`schedulable` - The schedulable object to make active.

getDefaultScheduler

`public static Scheduler`$_{54}$ **`getDefaultScheduler`**`()`

Gets a reference to the default scheduler.

Returns: A reference to the default scheduler.

getPolicyName

`public abstract java.lang.String getPolicyName()`

Gets a string representing the policy of `this`.

Returns: A `java.lang.String` object which is the name of the scheduling policy used by `this`.

isFeasible

`public abstract boolean isFeasible()`

Queries the system about the feasibility of the set of scheduling and release characteristics currently being considered.

removeFromFeasibility

```
protected abstract boolean removeFromFeasibility(Schedulable47
    schedulable)
```

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`63, `ReleaseParameters`66, `MemoryParameters`150, and `ProcessingGroupParameters`81, of this instance of `Schedulable`47 should no longer be considered in the feasibility analysis of the associated `Scheduler`54. Whether the resulting system is feasible or not, the removal is completed.

Returns: True, if the removal was successful. False, if the removal was unsuccessful.

setDefaultScheduler

`public static void setDefaultScheduler(Scheduler54 scheduler)`

Sets the default scheduler. This is the scheduler given to instances of `RealtimeThread`25 when they are constructed. The default scheduler is set to the required `PriorityScheduler`58 at startup.

Parameters:

`scheduler` - The `Scheduler` that becomes the default scheduler assigned to new threads. If null nothing happens.

setIfFeasible

```
public boolean setIfFeasible(Schedulable47 schedulable,
    ReleaseParameters66 release, MemoryParameters150 memory)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`schedulable` - The instance of `Schedulable`47 for which the changes are proposed.

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

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.

setIfFeasible

```
public boolean setIfFeasible(Schedulable47 schedulable,
    ReleaseParameters66 release, MemoryParameters150 memory,
    ProcessingGroupParameters81 group)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`schedulable` - The instance of `Schedulable`47 for which the changes are proposed.

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

`group` - The proposed processing group parameters.

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.

6.3 PriorityScheduler

Declaration

`public class **PriorityScheduler** extends Scheduler`[54]

Description

Class which represents the required (by the RTSJ) priority-based scheduler. The default instance is the required priority scheduler which does fixed priority, preemptive scheduling.

6.3.1 Fields

MAX_PRIORITY

`public static final int MAX_PRIORITY`

The maximum priority value used by the implmentation.

MIN_PRIORITY

`public static final int MIN_PRIORITY`

The minimum priority value used by the implmenetation.

6.3.2 Constructors

PriorityScheduler

`protected PriorityScheduler()`

Construct an instance of `PriorityScheduler`. Applications will likely not need any other instance other than the default instance.

6.3.3 Methods

addToFeasibility

`protected void addToFeasibility(Schedulable`[47] `schedulable)`

Inform `this` and cooperating facilities that the `ReleaseParameters`66 of the given instance of `Schedulable`47 should be considered in feasibility analysis until further notified.

Overrides: `addToFeasibility`55 in class `Scheduler`54

Parameters:

`schedulable` - The instance of `Schedulable`47 whose `ReleaseParameters`66 are to be added to the feasibility set.

fireSchedulable

`public void **fireSchedulable**(Schedulable47 schedulable)`

Triggers the execution of a `Schedulable`47 object (like an instance of `AsyncEventHandler`210).

Overrides: `fireSchedulable`55 in class `Scheduler`54

Parameters:

`schedulable` - The `Schedulable`47 object to make active.

getMaxPriority

`public int **getMaxPriority**()`

Gets the maximum priority available for a thread managed by this scheduler.

Returns: The value of the maximum priority.

getMaxPriority

`public static int **getMaxPriority**(java.lang.Thread thread)`

Gets the maximum priority of the given instance of `java.lang.Thread` . If the given thread is scheduled by the required `PriorityScheduler` the maximum priority of the `PriorityScheduler` is returned otherwise `Thread.MAX_PRIORITY` is returned.

Parameters:

`thread` - An instance of `java.lang.Thread` . If null, the maximum priority of the required `PriorityScheduler` is returned.

Returns: The maximum priority of the given instance of `java.lang.Thread` .

getMinPriority

`public int **getMinPriority**()`

Gets the minimum priority available for a thread managed by this scheduler.

Returns: The value of the minimum priority.

getMinPriority

`public static int getMinPriority(java.lang.Thread thread)`

Gets the minimum priority of the given instance of `java.lang.Thread` . If the given thread is scheduled by the required `PriorityScheduler` the minimum priority of the `PriorityScheduler` is returned otherwise `Thread.MIN_PRIORITY` is returned.

Parameters:

`thread` - An instance of `java.lang.Thread` . If null the minimum priority of the required `PriorityScheduler` is returned.

Returns: The value of the minimum priority of the given instance of `java.lang.Thread`

getNormPriority

`public int getNormPriority()`

Gets the normal priority available for a thread managed by this scheduler.

Returns: The value of the normal priority.

getNormPriority

`public static int getNormPriority(java.lang.Thread thread)`

Gets the minimum priority of the given instance of `java.lang.Thread` . If the given thread is scheduled by the required `PriorityScheduler` the normal priority of the `PriorityScheduler` is returned otherwise `Thread.NORM_PRIORITY` is returned.

Parameters:

`thread` - An instance of Thread. If null the normal priority of the required `PriorityScheduler` is returned.

Returns: The value of the normal priority of the given instance of `java.lang.Thread`

getPolicyName

`public java.lang.String getPolicyName()`

Gets the policy name of `this`.

Overrides: `getPolicyName`56 in class `Scheduler`54

Returns: The policy name (Fixed Priority) as a string.

instance

```
public static PriorityScheduler58 instance()
```

Return a reference to an instance of `PriorityScheduler`.

Returns: A reference to an instance of `PriorityScheduler`.

isFeasible

```
public boolean isFeasible()
```

Queries the system about the feasibility of the set of `Schedulable`47 objects with respect to their include in the feasibility set and the constraints expressed by their associated parameter objects.

Overrides: `isFeasible`56 in class `Scheduler`54

Returns: True if the system is able to satisfy the constraints expressed by the paramter object of all instance of `Schedulable`47 currently in the feasibility set. False if the system cannot satisfy those constraints.

removeFromFeasibility

```
protected boolean removeFromFeasibility(Schedulable47
      schedulable)
```

Inform `this` and cooperating facilities that the `ReleaseParameters`66 of the given instance of `Schedulable`47 should *not* be considered in feasibility analysis until further notified.

Overrides: `removeFromFeasibility`56 in class `Scheduler`54

Parameters:

`schedulable` - The instance of `Schedulable`47 whose `ReleaseParameters`66 are to be removed from the feasibility set.

Returns: True, if the removal was successful. False, if the removal was unsuccessful.

setIfFeasible

```
public boolean setIfFeasible(Schedulable47 schedulable,
      ReleaseParameters66 release, MemoryParameters150 memory)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Overrides: `setIfFeasible`$_{56}$ in class `Scheduler`$_{54}$

Parameters:

`schedulable` - The instance of `Schedulable`$_{47}$ for which the changes are proposed.

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

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.

setIfFeasible

```
public boolean setIfFeasible(Schedulable47 schedulable,
     ReleaseParameters66 release, MemoryParameters150 memory,
     ProcessingGroupParameters81 group)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Overrides: `setIfFeasible`$_{57}$ in class `Scheduler`$_{54}$

Parameters:

`schedulable` - The instance of `Schedulable`$_{47}$ for which the changes are proposed.

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

`group` - The proposed processing group parameters.

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.

6.4 SchedulingParameters

Declaration

`public abstract class` **`SchedulingParameters`**

Direct Known Subclasses: `PriorityParameters`$_{63}$

Description

Subclasses of `SchedulingParameters` (`PriorityParameters`$_{63}$, `ImportanceParameters`$_{65}$, and any others defined for particular schedulers) provide the parameters to be used by the `Scheduler`$_{54}$. Changes to the values in a parameters object affects the scheduling behaviour of all the `Schedulable`$_{47}$ 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

SchedulingParameters

`public` **`SchedulingParameters`**`()`

6.5 PriorityParameters

Declaration

`public class` **`PriorityParameters`** `extends` `SchedulingParameters`$_{63}$

Direct Known Subclasses: `ImportanceParameters`$_{65}$

Description

Instances of this class should be assigned to threads 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`58 is such a scheduler.

6.5.1 Constructors

PriorityParameters

`public` **`PriorityParameters`**`(int priority)`

Create an instance of `SchedulingParameters`63 with the given priority.

Parameters:

`priority` - The priority assigned to a thread. This value is used in place of the value returned by `java.lang.Thread.setPriority(int)` .

6.5.2 Methods

getPriority

`public int` **`getPriority`**`()`

Gets the priority value.

Returns: The priority.

setPriority

`public void` **`setPriority`**`(int priority)`
`throws IllegalArgumentException`

Set the priority value.

Parameters:

`priority` - The value to which priority is set.

Throws:

`java.lang.IllegalArgumentException` - Thrown if the given priority value is less than the minimum priority of the scheduler of any of the associated threads or greater then the maximum priority of the scheduler of any of the associated threads.

toString

`public java.lang.String` **`toString`**`()`

Converts the priority value to a string.

Overrides: `toString` in class `Object`

Returns: The string representing the value of priority.

6.6 ImportanceParameters

Declaration

`public class` **`ImportanceParameters`** `extends PriorityParameters`$_{63}$

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 `PriorityScheduler`$_{58}$ is not required to use importance. However, the RTSJ strongly suggests to implementers that a fairly simple subclass of `PriorityScheduler`$_{58}$ that uses importance can offer value to some real-time applications.

6.6.1 Constructors

ImportanceParameters

`public` **`ImportanceParameters`**`(int priority, int importance)`

Create an instance of `ImportanceParameters`.

Parameters:

`priority` - The priority assigned to an instance of `Schedulable`$_{47}$. This value is used in place of java.lang.Thread.priority.

`importance` - The importance value assigned to an instance of `Schedulable`$_{47}$.

6.6.2 Methods

getImportance

`public int` **`getImportance`**`()`

Gets the importance value.

Returns: The value of importance for the associated instance of `Schedulable`47.

setImportance

`public void **setImportance**(int importance)`

Sets the importance value.

toString

`public java.lang.String **toString**()`

Print the value of the importance value of the associated instance of `Schedulable`47

Overrides: `toString`64 in class `PriorityParameters`63

6.7 ReleaseParameters

Declaration

`public class **ReleaseParameters**`

Direct Known Subclasses: `AperiodicParameters`73, `PeriodicParameters`70

Description

The abstract top-level class for release characteristics of threads. 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.

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.

6.7.1 Constructors

ReleaseParameters

`protected` **`ReleaseParameters`**`()`

Create a new instance of `ReleaseParameters`.

ReleaseParameters

```
protected ReleaseParameters(RelativeTime182 cost,
    RelativeTime182 deadline,
    AsyncEventHandler210 overrunHandler,
    AsyncEventHandler210 missHandler)
```

Create a new instance of `ReleaseParameters` with the given parameter values.

Parameters:

`cost` - Processing time units per interval. 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 interval. 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. Equivalent to `RelativeTime(0,0)` if null.

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. Changing the deadline might not take effect after the expiration of the current deadline. More detail provided in the subclasses.

`overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds cost. Not required for minimum implementation. If null, nothing happens on the overrun condition, and `waitForNextPeriod` returns false immediately and updates the start time for the next period.

`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, nothing happens on the miss deadline condition.

6.7.2 Methods

getCost

`public RelativeTime`$_{182}$ **`getCost`**`()`

Gets the value of the cost field.

Returns: The value of cost.

getCostOverrunHandler

`public AsyncEventHandler`$_{210}$ **`getCostOverrunHandler`**`()`

Gets a reference to the cost overrun handler.

Returns: A reference to the associated cost overrun handler.

getDeadline

`public RelativeTime`$_{182}$ **`getDeadline`**`()`

Gets the value of the deadline field.

Returns: The value of the deadline.

getDeadlineMissHandler

`public AsyncEventHandler`$_{210}$ **`getDeadlineMissHandler`**`()`

Gets a reference to the deadline miss handler.

Returns: A reference to the deadline miss handler.

setCost

`public void` **`setCost`**`(RelativeTime`$_{182}$ `cost)`

Sets the cost value.

Parameters:

`cost` - Processing time units per period or per minimum interarrival interval. 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 period or per minimum interarrival interval. 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 period or interval. Equivalent to `RelativeTime(0,0)` if null.

setCostOverrunHandler

`public void` **`setCostOverrunHandler`**`(AsyncEventHandler`$_{210}$ `handler)`

Sets the cost overrun handler.

Parameters:

`handler` - This handler is invoked if an invocation of the schedulable object attempts to exceed `cost` time units in a period. Not required for minimum implementation. See comments in `setCost()`.

setDeadline

`public void` **`setDeadline`**`(RelativeTime`$_{182}$ `deadline)`

Sets the deadline value.

Parameters:

`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 period or minimum interarrival interval. Other implementations may use this parameter to compute execution eligibility.

setDeadlineMissHandler

`public void` **`setDeadlineMissHandler`**`(AsyncEventHandler`$_{210}$ `handler)`

Sets the deadline miss handler.

Parameters:

`handler` - This handler is invoked if the `run()` method of the schedulable object is still executing after the deadline has passed. Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate.

setIfFeasible

`public boolean` **`setIfFeasible`**`(RelativeTime`$_{182}$ `cost,`
    `RelativeTime`$_{182}$ `deadline)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`cost` - The proposed cost.

`deadline` - The proposed deadline.

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.

6.8 PeriodicParameters

Declaration

`public class` **`PeriodicParameters`** `extends ReleaseParameters`66

Description

This release parameter indicates that the `RealtimeThread.waitForNextPeriod()`38 method on the associated `Schedulable`47 object will be unblocked at the start of each period. When a reference to a `PeriodicParameters` object is given as a parameter to a constructor the `PeriodicParameters` object becomes bound to the object being created. Changes to the values in the `PeriodicParameters` object affect the constructed object. If given to more than one constructor 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.

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.

6.8.1 Constructors

PeriodicParameters

```
public PeriodicParameters(HighResolutionTime172 start,
        RelativeTime182 period, RelativeTime182 cost,
        RelativeTime182 deadline,
        AsyncEventHandler210 overrunHandler,
        AsyncEventHandler210 missHandler)
```

Create a `PeriodicParameters` object.

Parameters:

`start` - Time at which the first period begins. If a `RelativeTime`182, this time is relative to the first time the schedulable object becomes schedulable *(schedulable time)* (e.g., when `start()` is called on a thread). If an `AbsoluteTime`176 and it is before the schedulable time, start is equivalent to the schedulable time.

`period` - The period is the interval between successive unblocks of `RealtimeThread.waitForNextPeriod()`38. Must be greater than zero when entering feasibility analysis.

`cost` - Processing time per period. 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 period. 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 period. Equivalent to `RelativeTime(0,0)` if null.

`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 period. Other implementations may use this parameter to compute execution eligibility. If null, deadline will equal the period.

`overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds cost in the given period. Not required for minimum implementation. if null, nothing happens on the overrun condition.

`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, nothing happens on the miss deadline condition.

6.8.2 Methods

getPeriod

`public RelativeTime`$_{182}$ `getPeriod()`

Gets the period.

Returns: The current value in `period`.

getStart

`public HighResolutionTime`$_{172}$ `getStart()`

Gets the start time.

Returns: The current value in `start`.

setIfFeasible

`public boolean setIfFeasible(RelativeTime`$_{182}$ `period, RelativeTime`$_{182}$ `cost, RelativeTime`$_{182}$ `deadline)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Parameters:

`period` - The proposed period.

`cost` - The proposed cost.

`deadline` - The proposed deadline.

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.

setPeriod

```
public void setPeriod(RelativeTime182 p)
```

Sets the period.

Parameters:

`period` - The value to which `period` is set.

setStart

```
public void setStart(HighResolutionTime172 start)
```

Sets the start time.

Parameters:

`start` - The value to which `start` is set.

6.9 AperiodicParameters

Declaration

```
public class AperiodicParameters extends ReleaseParameters66
```

Direct Known Subclasses: `SporadicParameters75`

Description

This release parameter object characterizes a schedulable object that may become active at any time. When a reference to a `AperiodicParameters` object is given as a parameter to a constructor the `AperiodicParameters` object becomes bound to the object being created. Changes to the values in the `AperiodicParameters` object affect the constructed object. If given to more than one constructor 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.

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.

6.9.1 Constructors

AperiodicParameters

```
public AperiodicParameters(RelativeTime182 cost,
        RelativeTime182 deadline,
        AsyncEventHandler210 overrunHandler,
        AsyncEventHandler210 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. Equivalent to `RelativeTime`182 `(0,0)` if null.

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. Not used in feasibility analysis for minimum implementation. If null, the deadline will be `RelativeTime`182 `(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, nothing happens on the overrun condition.

`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, nothing happens on the miss deadline condition.

6.9.2 Methods

setIfFeasible

```
public boolean setIfFeasible(RelativeTime182 cost,
        RelativeTime182 deadline)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of Schedulable$_{47}$ as appropriate, with the new scheduling characteristics.

Overrides: setIfFeasible$_{69}$ in class ReleaseParameters$_{66}$

Parameters:

cost - The proposed cost. If zero, no change is made.

deadline - The proposed deadline. If zero, no change is made.

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.

6.10 SporadicParameters

Declaration

public class **SporadicParameters** extends AperiodicParameters$_{73}$

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 constructor, the SporadicParameters object becomes bound to the object being created. Changes to the values in the SporadicParameters object affect the constructed object. If given to more than one constructor, 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.

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.

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 RealtimeThread$_{25}$ the arrival time is the time at which the start() is invoked. For other instances of Schedulable$_{47}$ it may be required for the implmementation to save the arrival times. For instances of AsyncEventHandler$_{210}$ with a ReleaseParameters$_{66}$ type of SporadicParameters the implementation must maintain a queue of motonically increasing arrival times which correspond to the execution of the fire() method of the instance of AsyncEvent$_{207}$ bound to the instance of AsyncEventHandler$_{210}$.

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.

6.10.1 Fields

arrivalTimeQueueOverflowExcept

```
public static final java.lang.String
      arrivalTimeQueueOverflowExcept
```

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` then the `fire()` method shall throw a `ResourceLimitError`246. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler`210 is bound then the arrival time is ignored.

arrivalTimeQueueOverflowIgnore

```
public static final java.lang.String
      arrivalTimeQueueOverflowIgnore
```

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` then the arrival time is ignored.

arrivalTimeQueueOverflowReplace

```
public static final java.lang.String
      arrivalTimeQueueOverflowReplace
```

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` then the previous arrival time is overwritten by the new arrival time. However, the new time is adjusted so that the difference between it and the previous time is equal to the minimum interarrival time.

arrivalTimeQueueOverflowSave

```
public static final java.lang.String arrivalTimeQueueOverflowSave
```

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` then the queue is lengthened and the arrival time is saved.

mitViolationExcept

`public static final java.lang.String` **`mitViolationExcept`**

If an arrival time for any instance of `Schedulable`$_{47}$ which has `this` as its instance of `ReleaseParameters`$_{66}$ occurs at a time less then the minimum interarrival time defined here then the `fire()` method shall throw `MITViolationException`$_{252}$. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler`$_{210}$ is bound then the arrival time is ignored.

mitViolationIgnore

`public static final java.lang.String` **`mitViolationIgnore`**

If an arrival time for any instance of `Schedulable`$_{47}$ which has `this` as its instance of `ReleaseParameters`$_{66}$ occurs at a time less then the minimum interarrival time defined here then the new arrival time is ignored.

mitViolationReplace

`public static final java.lang.String` **`mitViolationReplace`**

If an arrival time for any instance of `Schedulable`$_{47}$ which has `this` as its instance of `ReleaseParameters`$_{66}$ occurs at a time less then the minimum interarrival time defined here then the previous arrival time is overwritten with the new arrival time.

mitViolationSave

`public static final java.lang.String` **`mitViolationSave`**

If an arrival time for any instance of `Schedulable`$_{47}$ which has `this` as its instance of `ReleaseParameters`$_{66}$ occurs at a time less then the minimum interarrival time defined here then the new arrival time is added to the queue of arrival times. However, the new time is adjusted so that the difference between it and the previous time is equal to the minimum interarrival time.

6.10.2 Constructors

SporadicParameters

```
public SporadicParameters(RelativeTime182 minInterarrival,
    RelativeTime182 cost, RelativeTime182 deadline,
AsyncEventHandler210 overrunHandler,
AsyncEventHandler210 missHandler)
```

Create a `SporadicParameters` object.

Parameters:

`minInterarrival` - The release times of the schedulable object will occur no closer than this interval. Must be greater than zero when entering feasibility analysis.

`cost` - Processing time per minimum interarrival interval. 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 interval. 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. Equivalent to `RelativeTime(0,0)` if null.

`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, deadline will equal the minimum interarrival time.

`overrunHandler` - This handler is invoked if an invocation of the schedulable object exceeds cost. Not required for minimum implementation. If null, nothing happens on the overrun condition.

`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, nothing happens on the miss deadline condition.

6.10.3 Methods

getArrivalTimeQueueOverflowBehavior

```
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.

getInitialArrivalTimeQueueLength

`public int getInitialArrivalTimeQueueLength()`

Gets the initial number of elements the arrival time queue can hold.

Returns: The initial length of the queue.

getMinimumInterarrival

`public RelativeTime182 getMinimumInterarrival()`

Gets the minimum interarrival time.

Returns: The minimum interarrival time.

getMitViolationBehavior

`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.

setArrivalTimeQueueOverflowBehavior

```
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.

Parameters:

`behavior` - A string representing the behavior.

setIfFeasible

```
public boolean setIfFeasible(RelativeTime182 interarrival,
    RelativeTime182 cost, RelativeTime182 deadline)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Parameters:

`interarival` - The proposed interarrival time.

`cost` - The proposed cost.

`deadline` - The proposed deadline.

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.

setInitialArrivalTimeQueueLength

`public void` **`setInitialArrivalTimeQueueLength`**`(int initial)`

Sets the initial number of elements the arrival time queue can hold without lengthening the queue.

Parameters:

`initial` - The initial length of the queue.

setMinimumInterarrival

`public void` **`setMinimumInterarrival`**`(RelativeTime`182 `minimum)`

Set the minimum interarrival time.

Parameters:

`minimum` - The release times of the schedulable object will occur no closer than this interval. Must be greater than zero when entering feasibility analysis.

setMitViolationBehavior

`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.

Parameters:

`behavior` - A string representing the behavior.

6.11 ProcessingGroupParameters

Declaration

`public class **ProcessingGroupParameters**`

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`. For all threads with a reference to an instance of `ProcessingGroupParameters` p and a reference to an instance of `AperiodicParameters`73 no more than `p.cost` will be allocated to the execution of these threads in each interval of time given by p.period after the time indicated by `p.start`. When a reference to a `ProcessingGroupParameters` object is given as a parameter to a constructor the `ProcessingGroupParameters` object becomes bound to the object being created. Changes to the values in the `ProcessingGroupParameters` object affect the constructed object. If given to more than one constructor, 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.

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.

6.11.1 Constructors

ProcessingGroupParameters

```
public ProcessingGroupParameters(HighResolutionTime172 start,
     RelativeTime182 period, RelativeTime182 cost,
     RelativeTime182 deadline,
     AsyncEventHandler210 overrunHandler,
     AsyncEventHandler210 missHandler)
```

Create a `ProcessingGroupParameters` object.

Parameters:

`start` - Time at which the first period begins.

`period` - The period is the interval between successive unblocks of `waitForNextPeriod()`.

`cost` - Processing time per period.

`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.

`overrunHandler` - This handler is invoked if the `run()` method of the schedulable object of the previous period is still executing at the start of the current period.

`missHandler` - This handler is invoked if the `run()` method of the schedulable object is still executing after the deadline has passed.

6.11.2 Methods

getCost

`public RelativeTime`$_{182}$ **`getCost`**`()`

Gets the the value of `cost`.

Returns: the value of `cost`.

getCostOverrunHandler

`public AsyncEventHandler`$_{210}$ **`getCostOverrunHandler`**`()`

Gets the cost overrun handler.

Returns: A reference to an instance of `AsyncEventHandler`$_{210}$ that is cost overrun handler of `this`.

getDeadline

`public RelativeTime`$_{182}$ **`getDeadline`**`()`

Gets the value of `deadline`.

Returns: A reference to an instance of `RelativeTime`$_{182}$ that is the deadline of `this`.

getDeadlineMissHandler

`public AsyncEventHandler`$_{210}$ **`getDeadlineMissHandler`**`()`

Gets the deadline miss handler.

Returns: A reference to an instance of `AsyncEventHandler`$_{210}$ that is deadline miss handler of `this`.

getPeriod

`public RelativeTime`$_{182}$ **`getPeriod`**`()`

Gets the value of `period`.

Returns: A reference to an instance of `RelativeTime`$_{182}$ that represents the value of `period`.

getStart

`public HighResolutionTime`$_{172}$ **`getStart`**`()`

Gets the value of `start`.

Returns: A reference to an instance of `HighResolutionTime`$_{172}$ that represents the value of `start`.

setCost

`public void` **`setCost`**`(RelativeTime`$_{182}$ `cost)`

Sets the value of `cost`.

Parameters:

`cost` - The new value for `cost`.

setCostOverrunHandler

`public void` **`setCostOverrunHandler`**`(AsyncEventHandler`$_{210}$ `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.

setDeadline

`public void` **`setDeadline`**`(RelativeTime`$_{182}$ `deadline)`

Sets the value of `deadline`.

Parameters:

`deadline` - The new valude for `deadline`.

setDeadlineMissHandler

`public void` **`setDeadlineMissHandler`**`(AsyncEventHandler`$_{210}$ `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.

setIfFeasible

`public boolean` **`setIfFeasible`**`(RelativeTime`$_{182}$ `period,`
`RelativeTime`$_{182}$ `cost, RelativeTime`$_{182}$ `deadline)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Parameters:

`period` - The proposed period.

`cost` - The proposed cost.

`deadline` - The proposed deadline.

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.

setPeriod

`public void` **`setPeriod`**`(RelativeTime`$_{182}$ `period)`

Sets the value of `period`.

Parameters:

`period` - The new value for `period`.

setStart

`public void` **`setStart`**`(HighResolutionTime`$_{172}$ `start)`

Sets the value of `start`.

Parameters:

`start` - The new value for `start`.

CHAPTER 7 Memory Management

This section contains classes that:

- 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 real-time threads.
- 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.

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.

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 execution 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) <= Cn for some constant C.
2. Execution time of object constructors is explicitly not considered in any bounds.
3. The structure of enclosing scopes is accessible through a set of methods on

RealtimeThread. These methods allow the outer scopes to be accessed like an array. The algorithms for maintaining the scope structure are given in “Maintaining the Scope Stack.”

4. A memory scope is represented by an instance of the `ScopedMemory` class. When a new scope is entered, by calling the `enter()` method of the instance or by starting an instance of `RealtimeThread` or `NoHeapRealtimeThread` whose constructors were given a reference to an instance of `ScopedMemory`, all subsequent uses of the `new` keyword within the program logic of the scope will allocate the memory from the memory represented by that instance of `ScopedMemory`. When the scope is exited by returning from the `enter()` method of the instance of `ScopedMemory`, all subsequent uses of the `new` operation will allocate the memory from the area of memory associated with the enclosing scope.
5. The parent of a scoped memory area is the memory area in which the object representing the scoped memory area is allocated.
6. The single parent rule requires that a scope memory area have exactly zero or one parent.
7. Memory scopes that are made current by entering them or passing them as the initial memory area for a new thread must satisfy the *single parent rule.*
8. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of threads in which it is being used.
9. When the reference count for an instance of the class `ScopedMemory` is decremented from one to zero, all objects within that area are considered unreachable and are candidates for reclamation. The finalizers for each object in the memory associated with an instance of `ScopedMemory` are executed to completion before any statement in any thread attempts to access the memory area again.
10. Objects created in any immortal memory area live for the duration of the application. Their finalizers are only run when the application is terminated.
11. The addresses of objects in any `MemoryArea` that is associated with a `NoHeapRealtimeThread` must remain fixed while they are alive.
12. Each instance of the virtual machine will have exactly one instance of the class `ImmortalMemory`.
13. Each instance of the virtual machine will have exactly one instance of the class `HeapMemory`.
14. Each instance of the virtual machine will behave as if there is an area of memory into which all `Class` objects are placed and which is unexceptionally referenceable by `NoHeapRealtimeThreads`.
15. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers, and thus maintain the pointer safety of Java.

The restrictions are listed in the following table:

	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Yes, if same, outer, or shared scope
Local Variable	Yes	Yes	Yes, if same, outer, or shared scope

16. 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).

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 method in MemoryArea, construction of a new `RealtimeThread`, the `executeInArea` method in `MemoryArea`, and all the new instance methods in `MemoryArea`.

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 goes to zero.
- 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)`:

```
if entering ma would violate the single parent rule
    throw ScopedCycleException
push ma on the scope stack belonging to the current thread
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 or immortal
    start a new scope stack containing only ma
    make the new scope stack the scope stack for the current threa
d
else ma is scoped
    if ma is in the scope stack for the current thread
        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 t
hread
    else
        throw InaccessibleAreaException
execute logic.run or construct the object
restore the previous scope stack for the current thread
discard the new scope stack
```

Construct a RealtimeThread

For construction of a `RealtimeThread` in memory area cma with initial memory area of ima:

```
if cma is heap or immortal
    create a new scope stack containing cma
else
    start a new scope stack containing the
        entire current scope stack
for every scoped memory area in the new scope stack
        increment the reference count
if ima != current allocation context
    push ima on new scope stack
        which may throw ScopedCycleException
run the new thread with the new scope stack
when the thread terminates
    every memory area pushed by the thread will have been popped
    for every scoped memory area in the scope stack
        decrement the reference count
    free the scope stack.
```

The Single Parent Rule

Every push of a scoped memory type on a scope stack requires reference to the single parent rule, which requires that every scoped memory area have no more than one parent.

The parent of a scoped memory area is (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 are below it on the scope stack.

Except for the *primordial scope,* which represents both heap and immortal memory, only scoped memory areas are visible to the single parent rule.

The operational effect of the single parent rule is that once a scoped memory area is assigned a parent none of the above operations can change the parent and 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 posible. Thus a thread 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 and immortal memory) at its root, and other nodes corresponding to ever scoped memory area that is currently on any threads 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.

On Scope Stack Push of ma
The following procedure could be used to maintain the scope tree and ensure that push operations on a process' scope stack do not violate the single parent rule.

```
precondition: ma.parent is set to the correct parent (either a s
coped memory area
    or the primordial area) or to noParent
    t.scopeStack is the scope stack of the current thread

if ma is scoped
   parent = findFirstScope(t.scopeStack)
   if ma.parent == noParent
       ma.parent = parent
   if ma.parent != parent
       throw ScopedCycleException
   else
       t.scopeStack.push(ma)
       ma.refCount++
```

`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 area
```

On Scope Stack Pop of ma

```
ma = t.scopeStack.pop()
if ma is scoped
    ma.refCount--
    if ma.refCount == 0
        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-determinancy they add to the execution of program logic. To date, the expert group believes that no garbage collector algorithm or implementation is known that allows preemption at points that leave the inter-object pointers in the heap in a consistent state and are sufficiently close in time to minimize the overhead added to task switch latencies to a sufficiently small enough value which could be considered appropriate for all real-time systems.

Thus, this specification provides the above described areas of memory to 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.

Illegal Parameters

Except as noted, all `byte`, `int`, and `long` parameter values documented in this chapter must be non-negative, and all object references must be non-null. The methods will thow an `IllegalArgumentException` if they are passed a negative integer-type parameter or a null object reference.

Many constructors for memory areas accept values for the area's initial size and its maximum size. These constructors must throw an `IllegalArgumentException` if the maximum size is less than the initial size.

7.1 MemoryArea

Declaration

`public abstract class` **`MemoryArea`**

Direct Known Subclasses: `HeapMemory`95, `ImmortalMemory`96, `ImmortalPhysicalMemory`113, `ScopedMemory`98

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.

7.1.1 Constructors

MemoryArea

`protected MemoryArea(long sizeInBytes)`

Create an instance of `MemoryArea`.

Parameters:

`sizeInBytes` - The size of `MemoryArea` to allocate, in bytes.

MemoryArea

`protected MemoryArea(long sizeInBytes, java.lang.Runnable logic)`

Create an instance of `MemoryArea`.

Parameters:

`sizeInBytes` - The size of `MemoryArea` to allocate, in bytes.

`logic` - The `run()` method of this object will be called whenever `enter()`92 is called.

MemoryArea

`protected MemoryArea(SizeEstimator97 size)`

Create an instance of `MemoryArea`.

Parameters:

`size` - A `SizeEstimator`97 object which indicates the amount of memory required by this `MemoryArea`.

MemoryArea

```
protected MemoryArea(SizeEstimator97 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()`92 is called.

7.1.2 Methods

enter

```
public void enter()
    throws ScopedCycleException
```

Associate this memory area to the current real-time thread for the duration of the execution of the `run()` method of the `java.lang.Runnable` passed at construction time or the current instance of `Schedulable`47. During this bound period of execution, all objects are allocated from the memory area until another one takes effect, or the `enter()` method is exited. A runtime exception is thrown if this method is called from thread other than a `RealtimeThread`25 or `NoHeapRealtimeThread`38.

Throws:

`java.lang.IllegalArgumentException` - Thrown if no `Runnable` was passed in the constructor.

`ScopedCycleException`253 - If entering this ScopedMemory would violate the single parent rule.

enter

```
public void enter(java.lang.Runnable logic)
    throws ScopedCycleException
```

Associate this memory area to the current real-time thread for the duration of the execution of the `run()` method of the given `java.lang.Runnable`. During this bound period of execution, all objects are allocated from the memory area until another one takes effect, or the `enter()` method is exited. A runtime exception is thrown if this method is called from thread other than a `RealtimeThread`25 or `NoHeapRealtimeThread`38.

Parameters:

`logic` - The Runnable object whose `run()` method should be invoked.

Throws:

`ScopedCycleException`253 - If entering this ScopedMemory would violate the single parent rule.

executeInArea

```
public void executeInArea(java.lang.Runnable logic)
      throws InaccessibleAreaException
```

Execute the run method from the `logic` parameter using this memory area as the current allocation context. If the memory area is a scoped memory type, this method behaves as if it had moved the allocation context up the scope stackto the occurance of the memory area. If the memory area is heap or immortal memory, this method behaves as if the run method were running in that memory type with an empty scope stack.

Parameters:

`logic` - The runnable object whose `run()` method should be executed.

Throws:

IllegalStateException - A non-realtime thread attempted to enter the memory area.

`InaccessibleAreaException`255 - The memory area is not in the thread's scope stack.

getMemoryArea

```
public static MemoryArea90 getMemoryArea(java.lang.Object object)
```

Gets the `MemoryArea` in which the given object is located.

Returns: The current instance of `MemoryArea` of the object.

memoryConsumed

```
public long memoryConsumed()
```

An exact count, in bytes, of the all of the memory currently used by the system for the allocated objects.

Returns: The amount of memory consumed in bytes.

memoryRemaining

```
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

newArray

```
public java.lang.Object newArray(java.lang.Class type,
      int number)
      throws IllegalAccessException, InstantiationException
```

Allocate an array of the given type in this memory area.

Parameters:

`type` - The class of the elements of the new array.

`number` - The number of elements in the new array.

Returns: A new array of class type, of number elements.

Throws:

`java.lang.IllegalAccessException` - The class or initializer is inaccessible.

`java.lang.InstantiationException` - The array cannot be instantiated.

OutOfMemoryError - Space in the memory area is exhausted.

newInstance

```
public java.lang.Object newInstance(java.lang.Class type)
      throws IllegalAccessException, InstantiationException
```

Allocate an object in this memory area.

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.InstantiationException` - 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.

OutOfMemoryError - Space in the memory area is exhausted.

newInstance

```
public java.lang.Object
     newInstance(java.lang.reflect.Constructor c,
     java.lang.Object[] args)
     throws IllegalAccessException, InstantiationException
```

Allocate an object in this memory area.

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.InstantiationException` - 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.

OutOfMemoryError - Space in the memory area is exhausted.

size

```
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 MemoryArea`90

Description

The `HeapMemory` class is a singleton object that allows logic within other memory areas to allocate objects in the Java heap.

7.2.1 Methods

instance

`public static HeapMemory`$_{95}$ **`instance()`**

Returns a pointer to the singleton instance of `HeapMemory` representing the Java heap.

Returns: The singleton `HeapMemory` object.

memoryConsumed

`public long` **`memoryConsumed()`**

Indicates the amount of memory currently allocated in the Java heap.

Overrides: `memoryConsumed`$_{93}$ in class `MemoryArea`$_{90}$

Returns: The amount of allocated memory in the Java heap in bytes.

memoryRemaining

`public long` **`memoryRemaining()`**

Indicates the free memory remaining in the Java heap.

Overrides: `memoryRemaining`$_{93}$ in class `MemoryArea`$_{90}$

Returns: The amount of free memory remaining in the Java heap in bytes.

7.3 ImmortalMemory

Declaration

`public final class` **`ImmortalMemory`** `extends MemoryArea`$_{90}$

Description

`ImmortalMemory` is a memory resource that is shared among all threads. Objects allocated in the immortal memory live until the end of the application. Objects in immortal memory are never subject to garbage collection, although some GC algorithms may require a scan of the immortal memory. An *immortal* object may only contain reference to other immortal objects or to heap objects. Unlike standard Java heap objects, immortal objects continue to exist even after there are no other references to them.

7.3.1 Methods

instance

`public static ImmortalMemory`$_{96}$ **`instance`**`()`

Returns a pointer to the singleton `ImmortalMemory` space.

Returns: The singleton `ImmortalMemory` object.

7.4 SizeEstimator

Declaration

`public final class` **`SizeEstimator`**

Description

This is a convenient class to help people figure out how much memory they need. Instead of passing actual numbers to the `MemoryArea`$_{90}$ constructors, one can pass `SizeEstimator` objects with which you can have a better feel of how big a memory area you require.

See Also: `MemoryArea.MemoryArea(SizeEstimator)`$_{91}$, `LTMemory.LTMemory(SizeEstimator, SizeEstimator)`$_{108}$, `VTMemory.VTMemory(SizeEstimator, SizeEstimator)`$_{106}$

7.4.1 Constructors

SizeEstimator

`public` **`SizeEstimator`**`()`

7.4.2 Methods

getEstimate

`public long` **`getEstimate`**`()`

Gets an estimate of the nummber of bytes needed to store all the all the objects reserved.

Returns: The estimated size in bytes.

reserve

`public void **reserve**(java.lang.Class clas, int number)`

Take into account additional `number` instances of Class `class` when estimating the size of the `MemoryArea`$_{90}$.

Parameters:

`clas` - The class to take into account.

`number` - The number of instances of `class` to estimate.

reserve

`public void **reserve**(SizeEstimator`$_{97}$ `size)`

Take into account an additional instance of SizeEstimator `size` when estimating the size of the `MemoryArea`$_{90}$.

Parameters:

`size` - The given instance of `SizeEstimator`.

reserve

`public void **reserve**(SizeEstimator`$_{97}$ `size, int number)`

Take into account additional `number` instances of SizeEstimator `size` when estimating the size of the `MemoryArea`$_{90}$.

Parameters:

`size` - The given instance of `SizeEstimator`.

`nnumber` - The number of instances of `class` to estimate.

7.5 ScopedMemory

Declaration

`public abstract class **ScopedMemory** extends MemoryArea`$_{90}$

Direct Known Subclasses: `LTMemory`$_{107}$, `LTPhysicalMemory`$_{122}$, `VTMemory`$_{105}$, `VTPhysicalMemory`$_{129}$

Description

`ScopedMemory` is the abstract base class of all classes dealing with representations of memory spaces which have a limited lifetime. The `ScopedMemory` area is valid as long as there are real-time threads with access to it. A reference is created for each accessor when either a real-time thread is created with the `ScopedMemory` object as its memory area, or a real-time thread runs the `enter()`$_{101}$ method for the memory area.

When the last reference to the object is removed, by exiting the thread or exiting the `enter()`101 method, finalizers are run for all objects in the memory area, and the area is emptied.

A `ScopedMemory` area is a connection to a particular region of memory and reflects the current status of it. 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 scheme in use, but the memory space that object represents is not. Typically, the memory for a `ScopedMemory` area might be allocated using native method implementations that make appropriate use of `malloc()` and `free()` or similar routines to manipulate memory.

The `enter()`101 method of `ScopedMemory` is the mechanism used to activate a new memory scope. Entry into the scope is done by calling the method:

```
public void enter(Runnable runnable)
```

Where `runnable` is a 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 when the `runnable.run()` method completes. Allocations of objects within `runnable.run()` are done with the `ScopedMemory` area. When `runnable.run()` is complete, the scoped memory area is no longer active. Its reference count will be decremented and if it is zero all of the objects in the memory area finalized and collected.

Objects allocated from a `ScopedMemory` area have a unique lifetime. They cease to exist on exiting a `enter()`101 method or upon exiting the last real-time thread referencing the area, regardless of any references that may exist to the object. Thus, to maintain the safety of Java and avoid dangling references, a very restrictive set of rules apply to `ScopedMemory` area objects:

1. A reference to an object in `ScopedMemory` can never be stored in an Object allocated in the Java heap.
2. A reference to an object in `ScopedMemory` can never be stored in an Object allocated in `ImmortalMemory`96 .
3. A reference to an object in `ScopedMemory` can only be stored in Objects allocated in the same `ScopedMemory` area, or into a—more inner—`ScopedMemory` area nested by the use of its `enter()` method.
4. References to immortal or heap objects *may* be stored into an object allocated in a `ScopedMemory` area.

7.5.1 Constructors

ScopedMemory

`public **ScopedMemory**(long size)`

Create a new `ScopedMemory` area with the given parameters.

Parameters:

`size` - The size of the new `ScopedMemory` area in bytes. If size is less than or equal to zero nothing happens.

ScopedMemory

`public **ScopedMemory**(long size, java.lang.Runnable runnable)`

Create a new `ScopedMemory` area with the given parameters.

Parameters:

`size` - The size of the new `ScopedMemory` area in bytes. If size is less than or equal to zero nothing happens.

ScopedMemory

`public **ScopedMemory**(SizeEstimator`$_{97}$ `size)`

Create a new `ScopedMemory` area with the given parameters.

Parameters:

`size` - The size of the new `ScopedMemory` area estimated by an instance of `SizeEstimator`$_{97}$.

ScopedMemory

`public **ScopedMemory**(SizeEstimator`$_{97}$ `size,`
`java.lang.Runnable runnable)`

Create a new `ScopedMemory` area with the given parameters.

Parameters:

`size` - The size of the new `ScopedMemory` area estimated by an instance of `SizeEstimator`$_{97}$.

`runnable` - The logic which will use the memory represented by `this` as its initial memory area.

7.5.2 Methods

enter

```
public void enter()
        throws ScopedCycleException
```

Associate this `ScopedMemory` area to the current realtime thread for the duration of the execution of the `run()` method of the current instance of `Schedulable`47 or the `run` method of the instance of `Schedulable`47 given in the constructor. During this bound period of execution, all objects are allocated from the `ScopedMemory` area until another one takes effect, or the `enter()` method is exited. A runtime exception is thrown if this method is called from a thread other than a `RealtimeThread`25 or `NoHeapRealtimeThread`38 .

Overrides: `enter`92 in class `MemoryArea`90

Throws:

`ScopedCycleException`253 - If the called from a thread which is not an instance of `Schedulable`47 .

enter

```
public void enter(java.lang.Runnable logic)
        throws ScopedCycleException
```

Associate this `ScopedMemory` area to the current realtime thread for the duration of the execution of the `run()` method of the given `java.lang.Runnable` . During this bound period of execution, all objects are allocated from the `ScopedMemory` area until another one takes effect, or the `enter()` method is exited. A runtime exception is thrown if this method is called from a thread other than a `RealtimeThread`25 or `NoHeapRealtimeThread`38 .

Overrides: `enter`92 in class `MemoryArea`90

Parameters:

`logic` - The runnable object which contains the code to execute.

Throws:

`ScopedCycleException`253 - If the called from a thread which is not an instance of `Schedulable`47 .

getMaximumSize

```
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.

getPortal

`public java.lang.Object` **`getPortal`**`()`

Return a reference to the portal object in this instance of `ScopedMemory`.

Returns: A reference to the portal object or null if there is no portal object.

getReferenceCount

`public int` **`getReferenceCount`**`()`

Returns the reference count of this `ScopedMemory`. The reference count is an indication of the number of threads that may have access to this scope.

Returns: The reference count of this `ScopedMemory`.

join

```
public void join()
    throws InterruptedException
```

Wait until the reference count of this `ScopedMemory` goes down to zero.

Throws:

`java.lang.InterruptedException` - If another thread interrupts this thread while it is waiting.

join

```
public void join(HighResolutionTime172 time)
    throws InterruptedException
```

Wait at most until the time designated by the `time` parameter for the reference count of this `ScopedMemory` to go down to zero.

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 another thread interrupts this thread while it is waiting.

joinAndEnter

```
public void joinAndEnter()
    throws InterruptedException, ScopedCycleException
```

Combine `join();enter();` such that no `enter()` from another thread 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, the method returns immediately.

Throws:

`java.lang.InterruptedException` - If another thread interrupts this thread while it is waiting.

`ScopedCycleException`253 - If entering this `ScopedMemory` would violate the single parent rule.

joinAndEnter

```
public void joinAndEnter(HighResolutionTime172 time)
    throws InterruptedException, ScopedCycleException
```

Combine `join(time);enter();` such that no `enter()` from another thread 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 at constructin time. If no `Runnable` was passed then this method returns immediately.

Parameters:

`time` - The time that bounds the wait.

Throws:

`java.lang.InterruptedException` - if another thread interrupts this thread while it is waiting.

`ScopedCycleException`253 - If entering this `ScopedMemory` would violate the single parent rule.

joinAndEnter

```
public void joinAndEnter(java.lang.Runnable logic)
     throws InterruptedException, ScopedCycleException
```

Combine `join();enter(logic);` such that no `enter` from another thread 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`

Parameters:

`logic` - The `java.lang.Runnable` object which contains the code to execute.

Throws:

`java.lang.InterruptedException` - If another thread interrupts this thread while it is waiting.

`ScopedCycleException`253 - If entering this `ScopedMemory` would violate the single parent rule.

joinAndEnter

```
public void joinAndEnter(java.lang.Runnable logic,
     HighResolutionTime172 time)
     throws InterruptedException, ScopedCycleException
```

Combine `join(time);enter(logic);` such that no `enter` from another thread 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`.

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 another thread interrupts this thread while it is waiting.

`ScopedCycleException`253 - If entering this `ScopedMemory` would violate the single parent rule.

setPortal

```
public void setPortal(java.lang.Object object)
```

Set the argument to the portal object in the memory area represented by this instance of `ScopedMemory`.

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`.

toString

`public java.lang.String toString()`

Returns a user-friendly representation of this `ScopedMemory`.

Overrides: `toString` in class `Object`

Returns: The string representation

7.6 VTMemory

Declaration

`public class VTMemory extends ScopedMemory`98

Description

The execution time of an allocation from a `VTMemory` area may take a variable amount of time. However, since `VTMemory` areas are not subject to garbage collection and objects within it may not be moved, these areas can be used by instances of `NoHeapRealtimeThread`38 .

7.6.1 Constructors

VTMemory

`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.

VTMemory

```
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.

VTMemory

```
public VTMemory(SizeEstimator97 initial, SizeEstimator97 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`97.

VTMemory

```
public VTMemory(SizeEstimator97 initial, SizeEstimator97 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`97.
- `logic` - An instance of `java.lang.Runnable` whose `run()` method will use `this` as its initial memory area.

7.6.2 Methods

getMaximumSize

`public long getMaximumSize()`

Gets the value of the maximum size to which `this` can grow.

Overrides: `getMaximumSize`101 in class `ScopedMemory`98

Returns: The maximum size value.

toString

`public java.lang.String toString()`

Create a string representing the name of `this`.

Overrides: `toString`105 in class `ScopedMemory`98

Returns: A string representing the name of `this`.

7.7 LTMemory

Declaration

`public class LTMemory extends ScopedMemory`98

Description

`LTMemory` represents a memory area, allocated per `RealtimeThread`25 , or for a group of real-time threads, guaranteed by the system to have linear time allocation. 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 memory area associated with a `NoHeapRealtimeThread`38 , or to enter the memory area using the `ScopedMemory.enter()`101 method within a `NoHeapRealtimeThread`38 . An `LTMemory` area has an initial size. Enough memory must be committed by the completion of the constructor to satisfy this initial 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 `LTMemory` with that initial memory size. *(Note: It is important to understand that the above statement does* ***not require that if the initial memory size is N and (sizeof(object_1) + sizeof(object_2) + ... + sizeof(object_n) = N) the allocations of objects 1 through N will necessarily***

succeed.) Execution time of an allocator allocating from this initial area must be linear in the size of the allocated object. Execution time of an allocator allocating from memory between initial and maximum is allowed to vary. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available. (Note: to ensure that all requested memory is available set initial and maximum to the same value)

See Also: `MemoryArea`*90*, `ScopedMemory`*98*, `RealtimeThread`*25*, `NoHeapRealtimeThread`*38*

7.7.1 Constructors

LTMemory

`public LTMemory(long initialSizeInBytes, long maxSizeInBytes)`

Create an `LTMemory` of the given size.

Parameters:

`initialSizeInBytes` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

`maxSizeInBytes` - The size in bytes of the memory to allocate for this area.

LTMemory

```
public LTMemory(long initialSizeInBytes, long maxSizeInBytes,
      java.lang.Runnable logic)
```

Create an `LTMemory` of the given size.

Parameters:

`initialSizeInBytes` - The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

`maxSizeInBytes` - 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.

LTMemory

`public LTMemory(SizeEstimator`*97* `initial, SizeEstimator`*97* `maximum)`

Create an `LTMemory` of the given size.

Parameters:

`initial` - An instance of `SizeEstimator`97 used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`maximum` - An instance of `SizeEstimator`97 used to give an estimate for the maximum bytes to allocate for this area.

LTMemory

```
public LTMemory(SizeEstimator97 initial, SizeEstimator97 maximum,
    java.lang.Runnable logic)
```

Create an `LTMemory` of the given size.

Parameters:

`initial` - An instance of `SizeEstimator`97 used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

`maximum` - An instance of `SizeEstimator`97 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.

7.7.2 Methods

getMaximumSize

```
public long getMaximumSize()
```

Gets the maximum allowable size for `this`.

Overrides: `getMaximumSize`101 in class `ScopedMemory`98

toString

```
public java.lang.String toString()
```

Prints the string “LTMemory”.

Overrides: `toString`105 in class `ScopedMemory`98

7.8 PhysicalMemoryManager

Declaration

`public final class` **`PhysicalMemoryManager`**

Description

The `PhysicalMemoryManager` is available for use by the various physical memory accessor objects (`VTPhysicalMemory`$_{129}$, `LTPhysicalMemory`$_{122}$, `ImmortalPhysicalMemory`$_{113}$, `RawMemoryAccess`$_{135}$, and `RawMemoryFloatAccess`$_{145}$) 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.

The base implementation will provide a `PhysicalMemoryManager` and a set of `PhysicalMemoryTypeFilter`$_{119}$ classes that correctly identify memory classes that are standard for the (OS, JVM, and processor) platform.

OEMs may provide `PhysicalMemoryTypeFilter`$_{119}$ 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 `MemoryTypeConflictException`$_{254}$, but since filters are not part of the normative RTSJ, this exception is at best advisory.

7.8.1 Fields

ALIGNED

`public static final java.lang.String` **`ALIGNED`**

Specify this to identify aligned memory.

BYTESWAP

`public static final java.lang.String` **`BYTESWAP`**

Specify this if byte swapping should be used.

DMA

`public static final java.lang.String` **`DMA`**

Specify this to identify DMA memory.

SHARED

`public static final java.lang.String` **`SHARED`**

Specify this to identify shared memory.

7.8.2 Constructors

PhysicalMemoryManager

`public` **`PhysicalMemoryManager`**`()`

7.8.3 Methods

isRemovable

`public static boolean` **`isRemovable`**`(long address, 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.

isRemoved

`public static boolean` **`isRemoved`**`(long address, 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.

onInsertion

`public static void` **`onInsertion`**`(long base, long size,`
    `AsyncEventHandler`$_{210}$ `aeh)`

Register the specified `AsyncEventHandler`210 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 any one of them. If the size or the base is less than 0, unregister all "remove" references to the handler.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

`aeh` - The handler to register.

Throws:

`java.lang.IllegalArgumentException` - If the specified range contains no removable memory.

onRemoval

```
public static void onRemoval(long base, long size,
        AsyncEventHandler210 aeh)
```

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 any one of them. If the size or the base is less than 0, remove all "remove" references to the handler parameter.

Parameters:

`base` - The starting address in physical memory.

`size` - The size of the memory area.

`aeh` - The handler to register.

Throws:

`java.lang.IllegalArgumentException` - if the specified range contains no removable memory.

registerFilter

```
public static final void registerFilter(java.lang.Object name,
        PhysicalMemoryTypeFilter119 filter)
        throws DuplicateFilterException, IllegalArgumentException
```

Register a memory type filter with the physical memory manager.

Parameters:

`name` - The type of memory handled by this filter.

`filter` - The filter object.

Throws:

`DuplicateFilterException`$_{251}$ - A filter for this type of memory already exists.

`java.lang.RuntimeException` - The system is configured for a bounded number of filters. This filter exceeds the bound.

`java.lang.IllegalArgumentException` - The name parameter must not be an array of objects.

`java.lang.IllegalArgumentException` - The name and filter must both be in immortal memory.

removeFilter

```
public static final void removeFilter(java.lang.Object name)
```

Remove the identified filter from the set of registered filters.

Parameters:

`name` - The identifying object for this memory attribute.

7.9 ImmortalPhysicalMemory

Declaration

```
public class ImmortalPhysicalMemory extends MemoryArea90
```

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 `ImmortalMemory`$_{96}$ memory areas, and may be used in any context where `ImmortalMemory` is appropriate. Objects allocated in immortal physical memory hava a lifetime greater than the application as do objects allocated in immortal memory.

7.9.1 Constructors

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long size)
    throws SecurityException, SizeOutOfBoundsException, Unsuppo
    rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long base,
     long size)
     throws SecurityException, SizeOutOfBoundsException, OffsetO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long base,
    long size, java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException, OffsetO
    utOfBoundsException, UnsupportedPhysicalMemoryException, Me
    moryTypeConflictException, MemoryInUseException
```

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.

`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()`92 is called.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long size,
     java.lang.Runnable logic)
     throws SecurityException, SizeOutOfBoundsException, Unsuppo
     rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`$_{92}$ is called.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{249}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{254}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long base,
     SizeEstimator97 size)
     throws SecurityException, SizeOutOfBoundsException, OffsetO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`*246* - The address is invalid.

`SizeOutOfBoundsException`*248* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*249* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*254* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`*254* - The specified memory is already in use.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type, long base,
    SizeEstimator97 size, java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException, OffsetO
    utOfBoundsException, UnsupportedPhysicalMemoryException, Me
    moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`*246* - The address is invalid.

`SizeOutOfBoundsException`*248* - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*249* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type,
      SizeEstimator97 size)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

ImmortalPhysicalMemory

```
public ImmortalPhysicalMemory(java.lang.Object type,
      SizeEstimator97 size, java.lang.Runnable logic)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

`logic` - The `run()` method of this object will be called whenever `MemoryArea.enter()`92 is called.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

7.10 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.

7.10.1 Methods

contains

`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.

find

`public long **find**(long base, long size)`

Search for memory of the right type.

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.

getVMAttributes

`public int **getVMAttributes**()`

Gets the virtual memory attributes of `this`.

Returns: The virtual memory attributes as an integer.

getVMFlags

`public int **getVMFlags**()`

Gets the virtual memory flags of `this`.

Returns: The virtual memory flags as an integer.

initialize

`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` - If the base and size do not fall into this type of memory.

isPrescent

`public boolean **isPresent**(long base, long size)`

Queries the system about the existance of 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.

isRemovable

`public boolean **isRemovable**()`

Queries the system about the removability of this memory.

Returns: true If this type of memory is removable.

onInsertion

```
public void onInsertion(long base, long size,
     AsyncEventHandler210 aeh)
```

Arrange for the specified AEH to be called if any memory in the specified range is inserted.

Parameters:

`base` - The physical address of the beginning of the memory region.

`size` - The size of the memory region.

`aeh` - Run the given handler if any memory in the specified range is removed.

Throws:

`java.lang.IllegalArgumentException` - If the base and size do not fall into this type of memory

onRemoval

```
public void onRemoval(long base, long size,
     AsyncEventHandler210 aeh)
```

Arrange for the specified AEH to be called if any memory in the specified range is removed.

Parameters:

`base` - The physical address of the beginning of the memory region.

`size` - the size of the memory region.

`aeh` - Run the given handler if any memory in the specified range is removed.

Throws:

`java.lang.IllegalArgumentException` - If the base and size do not fall into this type of memory

vFind

`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.

7.11 LTPhysicalMemory

Declaration

`public class LTPhysicalMemory extends ScopedMemory`$_{98}$

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 restrictive set of assignment rules as `ScopedMemory`$_{98}$ memory areas, and the same performance restrictions as `LTMemory`$_{107}$.

See Also: `MemoryArea`$_{90}$, `ScopedMemory`$_{98}$, `VTMemory`$_{105}$, `LTMemory`$_{107}$, `VTPhysicalMemory`$_{129}$, `ImmortalPhysicalMemory`$_{113}$, `RealtimeThread`$_{25}$, `NoHeapRealtimeThread`$_{38}$

7.11.1 Constructors

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long size)
     throws SecurityException, SizeOutOfBoundsException, Unsuppo
     rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long base,
     long size)
     throws SecurityException, SizeOutOfBoundsException, OffsetO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`*248* - The size is negative or extends into an invalid range of memory.

`OffsetOutOfBoundsException`*246* - The address is invalid.

`UnsupportedPhysicalMemoryException`*249* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*254* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`*254* - The specified memory is already in use.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long base,
    long size, java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException, OffsetO
    utOfBoundsException, UnsupportedPhysicalMemoryException, Me
    moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`SizeOutOfBoundsException`*248* - The size is negative or extends into an invalid range of memory.

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`*246* - The address is invalid.

UnsupportedPhysicalMemoryException$_{249}$ - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException$_{254}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

MemoryInUseException$_{254}$ - The specified memory is already in use.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long size,
     java.lang.Runnable logic)
     throws SecurityException, SizeOutOfBoundsException, Unsuppo
     rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - The size of the area in bytes.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException$_{248}$ - The size is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException$_{249}$ - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException$_{254}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long base,
     SizeEstimator97 size)
     throws SecurityException, SizeOutOfBoundsException, OffsetO
```

```
utOfBoundsException, UnsupportedPhysicalMemoryException, Me
moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type, long base,
      SizeEstimator97 size, java.lang.Runnable logic)
      throws SecurityException, SizeOutOfBoundsException, OffsetO
      utOfBoundsException, UnsupportedPhysicalMemoryException, Me
      moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type,
      SizeEstimator97 size)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

LTPhysicalMemory

```
public LTPhysicalMemory(java.lang.Object type,
      SizeEstimator97 size, java.lang.Runnable logic)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

7.11.2 Methods

toString

```
public java.lang.String toString()
```

Creates a string representing the name of `this`.

Overrides: `toString`105 in class `ScopedMemory`98

Returns: A string representing the name of `this`.

7.12 VTPhysicalMemory

Declaration

`public class` **`VTPhysicalMemory`** `extends ScopedMemory`98

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 restrictive set of assignment rules as `ScopedMemory`98 memory areas, and the same performance restrictions as `VTMemory`.

See Also: `MemoryArea`90, `ScopedMemory`98, `VTMemory`105, `LTMemory`107, `LTPhysicalMemory`122, `ImmortalPhysicalMemory`113, `RealtimeThread`25, `NoHeapRealtimeThread`38

7.12.1 Constructors

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long size)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.
- `size` - The size of the area in bytes.

Throws:

- `java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.
- `SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long base,
      long size)
      throws SecurityException, SizeOutOfBoundsException, OffsetO
      utOfBoundsException, UnsupportedPhysicalMemoryException, Me
      moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

SizeOutOfBoundsException248 - The size is negative or extends into an invalid range of memory.

OffsetOutOfBoundsException246 - The address is invalid.

UnsupportedPhysicalMemoryException249 - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

MemoryInUseException254 - The specified memory is already in use.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long base,
      long size, java.lang.Runnable logic)
      throws SecurityException, SizeOutOfBoundsException, OffsetO
      utOfBoundsException, UnsupportedPhysicalMemoryException, Me
      moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long size,
    java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException, Unsuppo
    rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - The size of the area in bytes.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long base,
     SizeEstimator97 size)
     throws SecurityException, SizeOutOfBoundsException, OffsetO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

MemoryInUseException254 - The specified memory is already in use.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type, long base,
    SizeEstimator97 size, java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException, OffsetO
    utOfBoundsException, UnsupportedPhysicalMemoryException, Me
    moryTypeConflictException, MemoryInUseException
```

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.

`base` - The physical memory address of the area.

`size` - A size estimator for this memory area.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

SizeOutOfBoundsException248 - The size extends into an invalid range of memory.

OffsetOutOfBoundsException246 - The address is invalid.

UnsupportedPhysicalMemoryException249 - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

MemoryInUseException254 - The specified memory is already in use.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type,
    SizeEstimator97 size)
    throws SecurityException, SizeOutOfBoundsException, Unsuppo
    rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

VTPhysicalMemory

```
public VTPhysicalMemory(java.lang.Object type,
      SizeEstimator97 size, java.lang.Runnable logic)
      throws SecurityException, SizeOutOfBoundsException, Unsuppo
      rtedPhysicalMemoryException, MemoryTypeConflictException
```

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.

`size` - A size estimator for this area.

`logic` - `enter` this memory area with this `Runnable` after the memory area is created.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`SizeOutOfBoundsException`248 - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{249}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{254}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

7.12.2 Methods

toString

`public java.lang.String toString()`

Creates a string representing the name of `this`.

Overrides: `toString`$_{105}$ in class `ScopedMemory`$_{98}$

Returns: A string representing the name of `this`.

7.13 RawMemoryAccess

Declaration

`public class RawMemoryAccess`

Direct Known Subclasses: `RawMemoryFloatAccess`$_{145}$

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 physicalarea 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 based on the value of the BYTE_ORDER static boolean variable in class `RealtimeSystem`$_{242}$.

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`$_{246}$. 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`$_{248}$. 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.

7.13.1 Constructors

RawMemoryAccess

```
public RawMemoryAccess(java.lang.Object type, long size)
     throws SecurityException, OffsetOutOfBoundsException, SizeO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException
```

Construct an instance of `RawMemoryAccess` with the given parameters.

Parameters:

`type` - An Object representing the type of memory required. Used to define the base address and control the mapping.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{249}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{254}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{254}$ - The specified memory is already in use.

RawMemoryAccess

```
public RawMemoryAccess(java.lang.Object type, long base,
     long size)
     throws SecurityException, OffsetOutOfBoundsException, SizeO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException
```

Construct an instance of `RawMemoryAccess` with the given parameters.

Parameters:

`type` - An Object representing the type of memory required. Used to define the base address and control the mapping.

`base` - The physical memory address of the region.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`254 - The specified memory is already in use.

7.13.2 Methods

getByte

```
public byte getByte(long offset)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the `byte` at the given offset.

Parameters:

`offset` - The offset at which to read the byte.

Returns: The byte read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

getBytes

```
public void getBytes(long offset, byte[] bytes, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` bytes starting at the given offset and assign them to the byte array passed starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`bytes` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

getInt

```
public int getInt(long offset)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the `int` at the given offset.

Parameters:

`offset` - The offset at which to read the integer.

Returns: The integer read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

getInts

```
public void getInts(long offset, int[] ints, int low, int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` integers starting at the given offset and assign them to the int array passed starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`ints` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

getLong

```
public long getLong(long offset)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the `long` at the given offset.

Parameters:

`offset` - The offset at which to read the long.

Returns: The long read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

getLongs

```
public void getLongs(long offset, long[] longs, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` longs starting at the given offset and assign them to the long array passed starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`longs` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException`*246* - The offset is invalid.

`SizeOutOfBoundsException`*248* - The size is negative or extends into an invalid address range.

getMappedAddress

```
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.

getShort

```
public short getShort(long offset)
      throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the `short` at the given offset.

Parameters:

`offset` - The offset at which to read the short.

Returns: The short read.

Throws:

`OffsetOutOfBoundsException`*246* - The offset is invalid.

`SizeOutOfBoundsException`*248* - The size is negative or extends into an invalid address range.

getShorts

```
public void getShorts(long offset, short[] shorts, int low,
      int number)
      throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` shorts starting at the given offset and assign them to the short array passed starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`shorts` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

map

```
public long map()
```

Maps the physical memory range into virtual memory. No-op if the system doesn't support virtual memory.

map

```
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.

Parameters:

`base` - The location to map at the virtual memory space.

Returns: The starting point of the virtual memory.

map

```
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.

Parameters:

`base` - The location to map at the virtual memory space.

`size` - Teh size of the block to map in.

Returns: The starting point of the virtual memory.

setByte

```
public void setByte(long offset, byte value)
        throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the `byte` at the given offset.

Parameters:

`offset` - The offset at which to write the byte.

`value` - The byte to write.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setBytes

```
public void setBytes(long offset, byte[] bytes, int low,
        int number)
        throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` bytes starting at the given offset from the byte array passed starting at position `low`.

Parameters:

`offset` - The offset at 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:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setInt

```
public void setInt(long offset, int value)
        throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the `int` at the given offset.

Parameters:

`offset` - The offset at which to write the integer.

`value` - The integer to write.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setInts

```
public void setInts(long offset, int[] ints, int low, int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` ints starting at the given offset from the int array passed starting at position `low`.

Parameters:

`offset` - The offset 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:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setLong

```
public void setLong(long offset, long value)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the `long` at the given offset.

Parameters:

`offset` - The offset at which to write the long.

`value` - The long to write.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setLongs

```
public void setLongs(long offset, long[] longs, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` longs starting at the given offset from the long array passed starting at position `low`.

Parameters:

`offset` - The offset 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:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setShort

```
public void setShort(long offset, short value)
      throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the `short` at the given offset.

Parameters:

`offset` - The offset at which to write the short.

`value` - The short to write.

Throws:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

setShorts

```
public void setShorts(long offset, short[] shorts, int low,
      int number)
      throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` shorts starting at the given offset from the short array passed starting at position `low`.

Parameters:

`offset` - The offset 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:

`OffsetOutOfBoundsException`246 - The offset is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid address range.

unmap

`public void` **`unmap`**`()`

Unmap the physical memory range from virtual memory. No-op if the system doesn't support virtual memory.

7.14 RawMemoryFloatAccess

Declaration

`public class` **`RawMemoryFloatAccess`** `extends RawMemoryAccess`135

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.

Many of the constructors and methods in this class throw `OffsetOutOfBoundsException`246 . 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`248 . 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

RawMemoryFloatAccess

```
public RawMemoryFloatAccess(java.lang.Object type, long size)
      throws SecurityException, OffsetOutOfBoundsException, SizeO
      utOfBoundsException, UnsupportedPhysicalMemoryException, Me
      moryTypeConflictException
```

Create a `RawMemoryFloatAccess` object using the given parameters.

Parameters:

`type` - An Object representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given type of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254

RawMemoryFloatAccess

```
public RawMemoryFloatAccess(java.lang.Object type, long base,
     long size)
     throws SecurityException, OffsetOutOfBoundsException, SizeO
     utOfBoundsException, UnsupportedPhysicalMemoryException, Me
     moryTypeConflictException
```

Create a `RawMemoryFloatAccess` object using the given parameters.

Parameters:

`type` - An Object representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping

`base` - The physical memory address of the area.

`size` - The size of the area in bytes.

Throws:

`java.lang.SecurityException` - The application doesn't have permissions to access physical memory or the given range of memory.

`OffsetOutOfBoundsException`246 - The address is invalid.

`SizeOutOfBoundsException`248 - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`249 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`254

7.14.2 Methods

getDouble

```
public double getDouble(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the double at the given `offset`.

Parameters:

`offset` - The offset at which to write the value.

Returns: The double value.

Throws:

`OffsetOutOfBoundsException` $_{246}$ - The address is invalid.

`SizeOutOfBoundsException` $_{248}$ - The size is negative or extends into an invalid range of memory.

getDoubles

```
public void getDoubles(long offset, double[] doubles, int low,
    int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` `double` values starting at the given `offset` in this, and assigns them into the `double` array starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`doubles` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException` $_{246}$ - The address is invalid.

`SizeOutOfBoundsException` $_{248}$ - The size is negative or extends into an invalid range of memory.

getFloat

```
public float getFloat(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets the float at the given `offset`.

Parameters:

`offsetThe` - offset at which to get the value.

Returns: The float value.

Throws:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

getFloats

```
public void getFloats(long offset, float[] floats, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Gets `number` `float` values starting at the given `offset` in this and assign them into the `byte` array starting at position `low`.

Parameters:

`offset` - The offset at which to start reading.

`floats` - The array into which the read items are placed.

`low` - The offset which is the starting point in the given array for the read items to be placed.

`number` - The number of items to read.

Throws:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

setDouble

```
public void setDouble(long offset, double value)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the double at the given `offset`.

Parameters:

`offset` - The offset at which to set the value.

`value` - The value which will be written.

Throws:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

setDoubles

```
public void setDoubles(long offset, double[] doubles, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` `double` values starting at the given `offset` in this, and assigns them into the `double` array starting at position `low`.

Parameters:

`offset` - The offset 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:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

setFloat

```
public void setFloat(long offset, float value)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets the float at the given `offset`.

Parameters:

`offset` - The offset at which to write the value.

`value` - The value which will be written.

Throws:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

setFloats

```
public void setFloats(long offset, float[] floats, int low,
     int number)
     throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

Sets `number` `float` values starting at the given `offset` in this from the `byte` array starting at position `low`.

Parameters:

`offset` - The offset 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 items to write.

Throws:

`OffsetOutOfBoundsException`$_{246}$ - The address is invalid.

`SizeOutOfBoundsException`$_{248}$ - The size is negative or extends into an invalid range of memory.

7.15 MemoryParameters

Declaration

`public class` **`MemoryParameters`**

Description

Memory parameters can be given on the constructor of `RealtimeThread`$_{25}$ and `AsyncEventHandler`$_{210}$. These can be used both for the purposes of admission control by the scheduler and for the purposes of pacing the garbage collector to satisfy all of the thread allocation rates.

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.

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

NO_MAX

```
public static final long NO_MAX
```

Specifies no maximum limit.

7.15.2 Constructors

MemoryParameters

```
public MemoryParameters(long maxMemoryArea, long maxImmortal)
    throws IllegalArgumentException
```

Create a `MemoryParameters` object with the given values.

Parameters:

`maxMemoryArea` - A limit on the amount of memory the thread may allocate in the memory area. Units are in bytes. If zero, no allocation allowed in the memory area. To specify no limit, use NO_MAX or a value less than zero.

`maxImmortal` - A limit on the amount of memory the thread may allocate in the immortal area. Units are in bytes. If zero, no allocation allowed in immortal. To specify no limit, use NO_MAX or a value less than zero.

Throws:

`java.lang.IllegalArgumentException`

MemoryParameters

```
public MemoryParameters(long maxMemoryArea, long maxImmortal,
    long allocationRate)
    throws IllegalArgumentException
```

Create a `MemoryParameters` object with the given values.

Parameters:

`maxMemoryArea` - A limit on the amount of memory the thread may allocate in the memory area. Units are in bytes. If zero, no allocation allowed in the memory area. To specify no limit, use NO_MAX or a value less than zero.

`maxImmortal` - A limit on the amount of memory the thread may allocate in the immortal area. Units are in bytes. If zero, no allocation allowed in immortal. To specify no limit, use NO_MAX or a value less than zero.

`allocationRate` - A limit on the rate of allocation in the heap. Units are in bytes per second. If zero, no allocation is allowed in the heap. To specify no limit, use NO_MAX or a value less than zero.

Throws:

`java.lang.IllegalArgumentException`

7.15.3 Methods

getAllocationRate

`public long` **`getAllocationRate`**`()`

Gets the allocation rate. Units are in bytes per second.

Returns: The allocation in bytes per second.

getMaxImmortal

`public long` **`getMaxImmortal`**`()`

Gets the limit on the amount of memory the thread may allocate in the immortal area. Units are in bytes.

Returns: The the limit of immortal memory.

getMaxMemoryArea

`public long` **`getMaxMemoryArea`**`()`

Gets the limit on the amount of memory the thread may allocate in the memory area. Units are in bytes.

Returns: The the allocation limit in this area.

setAllocationRate

`public void` **`setAllocationRate`**`(long allocationRate)`

Sets the limit on the rate of allocation in the heap.

Parameters:

`allocationRate` - Units are in bytes per second. If zero, no allocation is allowed in the heap. To specify no limit, use NO_MAX or a value less than zero.

setAllocationRateIfFeasible

`public boolean **setAllocationRateIfFeasible**(int allocationRate)`

Sets the limit on the rate of allocation in the heap. If this `Memory-Parameters` object is currently associated with one or more realtime threads 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. If zero, no allocation is allowed in the heap. To specify no limit, use NO_MAX or a value less than zero.

Returns: True if the request was fulfilled.

setMaxImmortalIfFeasible

`public boolean **setMaxImmortalIfFeasible**(long maximum)`

Sets the limit on the amount of memory the thread 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 or a value less than zero.

Returns: True if the value is set. False if any of the threads have already allocated more than the given value. In this case the call has no effect.

setMaxMemoryAreaIfFeasible

`public boolean **setMaxMemoryAreaIfFeasible**(long maximum)`

Sets the limit on the amount of memory the thread may allocate in the memory area.

Parameters:

`maximum` - Units are in bytes. If zero, no allocation allowed in the memory area. To specify no limit, use NO_MAX or a value less than zero.

Returns: True if the value is set. False if any of the threads have already allocated more than the given value. In this case the call has no effect.

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. In addition, the method(s) in `GarbageCollector` shall be made available by all implementations.

7.16.1 Constructors

GarbageCollector

`public` **`GarbageCollector`**`()`

Create an instance of `this`.

7.16.2 Methods

getPreemptionLatency

`public abstract RelativeTime`$_{182}$ **`getPreemptionLatency`**`()`

Preemption latency is a measure of the maximum time a `RealtimeThread`$_{25}$ may have to wait for the collector to reach a preemption-safe point. Instances of `RealtimeThread`$_{25}$ are allowed to preeempt the garbage collector (instances of `NoHeapRealtimeThread`$_{38}$ preempt immediately but instances of `RealtimeThread`$_{25}$ must wait until the collector reaches a preemption-safe point).

Returns: The preempting latency of the garbage collection algorithm represented by `this` if applicable. May return 0if there is no collector available.

CHAPTER 8

Synchronization

This section contains classes that:

- Allow the application of the priority ceiling emulation algorithm to individual objects.
- Allow the setting of the system default priority inversion algorithm.
- Allow wait-free communication between real-time threads and regular Java threads.

The specification strengthens the semantics of Java synchronization for use in real-time systems by mandating monitor execution eligibility control, commonly referred to as priority inversion control. A `MonitorControl` class is defined as the superclass of all such execution eligibility control algorithms. `PriorityInheritance` is the default monitor control policy; the specification also defines a `PriorityCeilingEmulation` option.

The wait-free queue classes provide protected, concurrent access to data shared between instances of `java.lang.Thread` and `NoHeapRealtimeThread`.

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. Threads waiting to enter synchronized blocks are priority queue ordered. If threads with the same priority are possible under the active scheduling policy such threads are queued in FIFO order.
2. Any conforming implementation must provide an implementation of the synchro-

nized 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.

3. The Priority Inheritance monitor control policy must be implemented.
4. Implementations that provide a monitor control algorithm in addition to those described herein are required to clearly document the behavior of that algorithm.

Rationale

Java monitors, and especially the synchronized keyword, provide a very elegant means for mutual exclusion synchronization. Thus, rather than invent a new real-time synchronization mechanism, this specification strengthens the semantics of Java synchronization to allow its use in real-time systems. In particular, this specification mandates priority inversion control. Priority inheritance and priority ceiling emulation are both popular priority inversion control mechanisms; however, priority inheritance is more widely implemented in real-time operating systems and so is the default mechanism in this specification.

By design the only mechanism required by this specification which can enforce mutual exclusion in the traditional sense is the keyword `synchronized`. Noting that the specification allows the use of `synchronized` by both instances of `java.lang.Thread`, `RealtimeThread`, and `NoHeapRealtimeThread` and that such flexibility precludes the correct implementation of *any* known priority inversion algorithm when locked objects are accessed by instances of `java.lang.Thread` and `NoHeapRealtimeThread`, it is incumbent on the specification to provide alternate means for protected, concurrent data access by both types of threads (protected means access to data without the possibility of corruption). The three wait-free queue classes provide such access.

8.1 MonitorControl

Declaration

`public abstract class` **`MonitorControl`**

Direct Known Subclasses: `PriorityCeilingEmulation`$_{158}$, `PriorityInheritance`$_{158}$

Description

Abstract superclass for all monitor control policy objects.

8.1.1 Constructors

MonitorControl

`public **MonitorControl**()`

Create an instance of `MonitorControl`.

8.1.2 Methods

getMonitorControl

`public static MonitorControl`$_{156}$ `**getMonitorControl**()`

Gets the monitor control policy of the given instance of `this`.

Returns: The monitor control policy object.

getMonitorControl

`public static MonitorControl`$_{156}$
`**getMonitorControl**(java.lang.Object monitor)`

Gets the monitor control policy of the given instance of `java.lang.Object` .

Returns: The monitor control policy object.

setMonitorControl

`public static void **setMonitorControl**(MonitorControl`$_{156}$ `policy)`

Sets the default monitor behavior for object monitors used by synchronized statements and methods in the system. The type of the policy object determines the type of behavior. Conforming implementations must support priority ceiling emulation and priority inheritance for fixed priority preemptive threads.

Parameters:

`policy` - The new monitor control policy. If null nothing happens.

setMonitorControl

`public static void **setMonitorControl**(java.lang.Object monitor,`
`MonitorControl`$_{156}$ `policy)`

Has the same effect as `setMonitorControl()`, except that the policy only affects the indicated object monitor.

Parameters:

`monitor` - The monitor for which the new policy will be in use. The policy will take effect on the first attempt to lock the monitor after the completion of this method. If null nothing will happen.

`policy` - The new policy for the object. If null nothing will happen.

8.2 PriorityCeilingEmulation

Declaration

`public class` **`PriorityCeilingEmulation`** `extends MonitorControl`156

Description

Monitor control class specifying use of the priority ceiling emulation protocol for monitor objects. Objects under the influence of this protocol have the effect that a thread entering the monitor has its effective priority—for priority-based dispatching—raised to the ceiling on entry, and is restored to its previous effective priority when it exits the monitor. See also `MonitorControl`156 and `PriorityInheritance`158 .

8.2.1 Constructors

PriorityCeilingEmulation

`public` **`PriorityCeilingEmulation`**`(int ceiling)`

Create a `PriorityCeilingEmulation` object with a given ceiling.

Parameters:

`ceiling` - Priority ceiling value.

8.2.2 Methods

getDefaultCeiling

`public int` **`getDefaultCeiling`**`()`

Gets the priority ceiling for this `PriorityCeilingEmulation` object.

Returns: The priority ceiling.

8.3 PriorityInheritance

Declaration

`public class` **`PriorityInheritance`** `extends MonitorControl`156

Description

Monitor control class specifying use of the priority inheritance protocol for object monitors. Objects under the influence of this protocol have the effect that a thread entering the monitor will boost the effective priority of the thread in the monitor to its own effective priority. When that thread exits the monitor, its effective priority will be restored to its previous value. See also `MonitorControl`156 and `PriorityCeilingEmulation`158

8.3.1 Constructors

PriorityInheritance

```
public PriorityInheritance()
```

8.3.2 Methods

instance

```
public static PriorityInheritance158 instance()
```

Return a pointer to the singleton `PriorityInheritance`.

8.4 WaitFreeWriteQueue

Declaration

```
public class WaitFreeWriteQueue
```

Description

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread`25 and `java.lang.Thread` . The problem is that synchronized access objects shared between real-time threads and threads might cause the real-time threads to incur delays due to execution of the garbage collector.

The `write` method of this class does not block on an imagined queue-full condition variable. If the `write()` method is called on a full queue false is returned. If two real-time threads intend to read from this queue they must provide their own synchronization.

The `read()` method of this queue is synchronized and may be called by more than one writer and will block on queue empty.

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 written to the previous constructor. The fix is to remove the exceptions from the catch clause around the constructor.

8.4.1 Constructors

WaitFreeWriteQueue

```
public WaitFreeWriteQueue(java.lang.Thread writer,
     java.lang.Thread reader, int maximum, MemoryArea90 memory)
     throws IllegalArgumentException
```

A queue with an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

Parameters:

`writer` - An instance of `java.lang.Thread` .

`reader` - An instance of `java.lang.Thread` .

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea`$_{90}$ in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - If an argument holds an invalid value. The current memory areas of `writer`, `reader`, and `memory` must be compatible with respect to the assignment and access rules for memory areas.

8.4.2 Methods

clear

```
public void clear()
```

Sets `this` to empty.

force

```
public boolean force(java.lang.Object object)
     throws MemoryScopeException
```

Force this `java.lang.Object` to replace the last one. If the reader should happen to have just removed the other `java.lang.Object` just as we were updating it, we will return false. False may mean that it just saw what we put in there. Either way, the best thing to do is to just write again—which will succeed, and check on the readers side for consecutive identical read values.

Returns: True, if an element was overwritten. False, if there as an empty element into which the write occurred.

Throws:

`MemoryScopeException`250

isEmpty

`public boolean isEmpty()`

Queries the system to determine if `this` is empty.

Returns: True, if `this` is empty. False, if `this` is not empty.

isFull

`public boolean isFull()`

Queries the system to determine if `this` is full.

Returns: True, if `this` is full. False, if `this` is not full.

read

`public java.lang.Object read()`

A synchronized read on the queue.

Returns: The `java.lang.Object` read or null if `this` is empty.

size

`public int size()`

Queries the system to determine the number of elements in this.

Returns: An integer which is the number of non-empty positions in `this`.

write

```
public boolean write(java.lang.Object object)
    throws MemoryScopeException
```

Attempt to insert an element into the queue.

Parameters:

`object` - The `java.lang.Object` to insert.

Returns: True, if the write succeeded. False, if not.

Throws:

`MemoryScopeException`$_{250}$

8.5 WaitFreeReadQueue

Declaration

`public class` **`WaitFreeReadQueue`**

Description

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread`$_{25}$ and `java.lang.Thread` . The problem is that synchronized access objects shared between real-time threads and threads might cause the real-time threads to incur delays due to execution of the garbage collector.

The `read()` method of this class does not block on an imagined queue-empty condition variable. If the `read()` is called on an empty queue null is returned. If two real-time threads intend to read from this queue they must provide their own synchronization.

The write method of this queue is synchronized and may be called by more than one writer and will block on queue empty.

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 written to the previous constructor. The fix is to remove the exceptions from the catch clause around the constructor.

8.5.1 Constructors

WaitFreeReadQueue

```
public WaitFreeReadQueue(java.lang.Thread writer,
      java.lang.Thread reader, int maximum, MemoryArea90 memory)
      throws IllegalArgumentException
```

A queue with an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Parameters:

`writer` - An instance of `java.lang.Thread` .

`reader` - An instance of `java.lang.Thread` .

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea`*90* in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - If an argument holds an invalid value. The current memory areas of writer, reader, and memory must be compatible with respect to the assignment and access rules for memory areas.

WaitFreeReadQueue

```
public WaitFreeReadQueue(java.lang.Thread writer,
     java.lang.Thread reader, int maximum, MemoryArea90 memory,
     boolean notify)
     throws IllegalArgumentException
```

A queue with an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Parameters:

`writer` - An instance of `java.lang.Thread` .

`reader` - An instance of `java.lang.Thread` .

`maximum` - The maximum number of elements in the queue.

`memory` - The `MemoryArea`*90* in which this object and internal elements are allocated.

`notify` - Whether or not the reader is notified when data is added.

Throws:

`java.lang.IllegalArgumentException` - If an argument holds an invalid value. The current memory areas of `writer`, `reader`, and `memory` must be compatible with respect to the assignment and access rules for memory areas.

8.5.2 Methods

clear

```
public void clear()
```

Sets `this` to empty.

isEmpty

```
public boolean isEmpty()
```

Queries the system to determine if `this` is empty.

Returns: True, if `this` is empty. False, if `this` is not empty.

isFull

`public boolean **isFull**()`

Queries the system to determine if `this` is full.

Returns: True, if `this` is full. False, if `this` is not full.

read

`public java.lang.Object **read**()`

Reads the next element in the queue unless the queue is empty. If the queue is empty null is returned.

Returns: The instance of `java.lang.Object` read. Null, if `this` was empty.

size

`public int **size**()`

Queries the system to determine the number of elements in this.

Returns: An integer which is the number of non-empty positions in `this`.

waitForData

`public void **waitForData**()`

If `this` is empty `waitForData()` waits on the event until the writer inserts data. Note that true priority inversion does not occur since the writer locks a different object and the notify is executed by the `AsyncEventHandler`*210* which has `noHeap` characteristics.

write

```
public boolean write(java.lang.Object object)
       throws MemoryScopeException
```

The 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 this.

Returns: True, if the write occurred. False, if it did not.

Throws:

MemoryScopeException$_{250}$

8.6 WaitFreeDequeue

Declaration

public class **WaitFreeDequeue**

Description

The wait-free queue classes facilitate communication and synchronization between instances of RealtimeThread$_{25}$ and `java.lang.Thread` . See WaitFreeWriteQueue$_{159}$ or WaitFreeReadQueue$_{162}$ for more details. Instances of this class create a WaitFreeWriteQueue$_{159}$ and a WaitFreeReadQueue$_{162}$ and make calls on the respective `read()` and `write()` methods.

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 written to the previous constructor. The fix is to remove the exceptions from the catch clause around the constructor.

8.6.1 Constructors

WaitFreeDequeue

```
public WaitFreeDequeue(java.lang.Thread writer,
        java.lang.Thread reader, int maximum, MemoryArea90 area)
        throws IllegalArgumentException
```

A queue with unsynchronized and nonblocking `read()` and `write()` methods and synchronized and blocking `read()`and `write()` methods.

Parameters:

`writer` - An instance of Thread.

`reader` - An instance of Thread.

`maximum` - Then maximum number of elements in the both the WaitFreeReadQueue$_{162}$ and the WaitFreeWriteQueue$_{159}$.

`area` - The MemoryArea$_{90}$ in which this object and internal elements are allocated.

Throws:

`java.lang.IllegalArgumentException` - If an argument holds an invalid value. The current memory areas of `writer`, `reader`, and `memory` must be compatible with respect to the assignment and access rules for memory areas.

8.6.2 Methods

blockingRead

`public java.lang.Object` **`blockingRead`**`()`

A synchronized call of the `read()` method of the underlying `WaitFreeWriteQueue`159. 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.

blockingWrite

`public boolean` **`blockingWrite`**`(java.lang.Object object)`
`throws MemoryScopeException`

A synchronized call of the `write()` method of the underlying `WaitFreeReadQueue`162. This call blocks on queue full and waits until there is space in this.

Parameters:

`object` - The `java.lang.Object` to place in this.

Returns: True, if the write succeeded. False, if not.

Throws:

`MemoryScopeException`250 - If the write causes an access or assignment violation.

force

`public boolean` **`force`**`(java.lang.Object object)`

If this is full then this call overwrites the last object written to this with the given object. If this is not full this call is equivalent to the `nonBlocking-Write()` call.

Parameters:

`object` - The `java.lang.Object` which will overwrite the last object if this is full. Otherwise `object` will be placed in this.

Returns: True, if an element was overwritten. False, if there as an empty element into which the write occurred.

nonBlockingRead

`public java.lang.Object` **`nonBlockingRead`**`()`

An unsynchronized call of the `read()` method of the underlying `WaitFreeReadQueue`162 .

Returns: A `java.lang.Object` object read from this. If there are no elements in this then null is returned.

nonBlockingWrite

```
public boolean nonBlockingWrite(java.lang.Object object)
      throws MemoryScopeException
```

An unsynchronized call of the `write()` method of the underlying `WaitFreeWriteQueue`159 . This call does not block on queue full.

Parameters:

`object` - The `java.lang.Object` to attempt to place in this.

Returns: True if the `object` is now in this, otherwise returns false.

Throws:

`MemoryScopeException`250 - If the write causes an access or assignment violation.

CHAPTER 9 Time

This section contains classes that:

- 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, times relative to some starting point, and a new construct, rational time, which allows the efficient expression of occurrences per some interval of relative time.

The time classes required by the specification are `HighResolutionTime`, `AbsoluteTime, RelativeTime`, and `RationalTime.`

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 expressed relative to midnight January 1, 1970 GMT. An instance of `RelativeTime` encapsulates a point in time that is relative to some other time value. Instances of `RationalTime` express a frequency by a numerator of type `long` (the frequency) and a denominator of type `RelativeTime.` If instances of `RationalTime` are given to certain constructors or methods the activity occurs for frequency times every interval. For example, if a `PeriodicTimer` is given an instance of `RationalTime` of (29,232) then the system will guarantee that the timer will fire exactly 29 times every 232 milliseconds even if the system has to slightly adjust the time between firings.

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, or from millisecond/ nanosecond values.
3. Time objects must provide simple addition and subtraction operations, both for the entire object and for constituent parts.
4. Time objects must implement the `Comparable` interface if it is available. The `compareTo()` method must be implemented even if the interface is not available.
5. ~~Any method of constructor that accepts a RationalTime of (x,y) must guarantee that its activity occurs exactly x times in every y milliseconds even if the intervals between occurrences of the activity have to be adjusted slightly. The RTSJ does not impose any required distribution on the lengths of the intervals but strongly suggests that implementations attempt to make them of approximately equal lengths.~~
6. The three items below are new and replace the item above. They are not underlined because other formatting is lost by underlining.
7. An instance of `RationalTime` is always interpreted as an interval of time (just as is an instance of `RelativeTime`). The interval, identified as the *subinterval* in items 6-8 below, is given by $(((millis/10^3)+(nanos/10^9))/frequency)$ seconds.
8. Methods or constructors *use* instances of `RationalTime` in one of two ways, as a single use or as a repetitive use (it is always clear from the context which use is appropriate). When used singly, the interval is calculated as above but used as the nearest expressible value, rounded down. E.g., the `RationalTime` instance of value, (frequency = 3, interval = (1000 milliseconds, 0 nanoseconds)) used singly must be interpreted as an interval of length 333 milliseconds plus 333,333 nanoseconds, which is 333,333,333 nanoseconds. (The nearest expressible value with respect to the fields in the class `RationalTime` and the values expressible by their types as defined by The Java Language Specification).
9. A repetitive use of an instance of `RationalTime` first establishes a framing which is an implicit and infinite sequence of times separated by, exactly, the subinterval $(((millis/10^3)+(nanos/10^9))/frequency)$ seconds and starting at a given start time. With infinite precision and expressibility the activity would occur at each framing time. However, since the subinterval, by design, may not be expressible exactly, the RTJS requires the following, less rigorous, guarantee; the method's or constructor's activity must occur exactly `frequency` times in each

interval of length `((millis/10`^{3}`) + (nanos/10`^{9}`))` seconds, even if the subintervals between occurrences of the activity must be of unequal lengths due to expressibility limitations. E.g., the `RationalTime` instance of value, (frequency = 3, interval = (1000 milliseconds, 0 nanoseconds)) used repetitively must be interpreted as an interval of length exactly `1/3` seconds, thus the framing times are, assuming a starting time of `x, (x+i/3, ... (i = 1..infinity))`. However the RTSJ considers an implementation conformant if it causes the activity to occur at of each of the following times, `(x+i.0, x+i.333333333, x+i.666666667, ... (i = 0..infinity))`. Indeed, the implementation would be considered conformant it the activity occurred at times, `(x+i.0, x+i.000000001, x+i.000000002, ... (i = 0..infinity)).`, which, although may seem odd, might meet some cost/quality metric for a particular market (e.g., children's toys). The RTSJ does not impose any required distribution on the relationship of the lengths of the subintervals but strongly suggests that implementations attempt to make them of approximately equal lengths, within expressibility limits.

10. The requirement of item 7 above must hold even given the presence of finite Clock precision, Clock jitter, and Clock cumulative inaccuracy. (Implementation note: Of the three above problems, precision and jitter can be known in all cases and cumulative inaccuracy can only be known for software clocks. However the RTSJ allows the implementation to assume that the associated hardware clock has no cumulative inaccuracies (because such a cumulative inaccuracy cannot be known except to compare the clock to an external timekeeping mechanism. But, such a comparison in itself suffers from relativistic problems and theory tells us it is impossible to do correctly to arbitrary precision). These three effects must be corrected in the repetitive use of an instance of `RationalTime` so that an *imagined* external timekeeping mechanism with exactly the same cumulative inaccuracy of the internal hardware clock and infinite precision will detect exactly `frequency` occurrences of the activity in each interval as defined above.

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 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
```

All Implemented Interfaces: `java.lang.Comparable`

Direct Known Subclasses: `AbsoluteTime`176, `RelativeTime`182

Description

Class `HighResolutionTime` is the base class for AbsoluteTime, RelativeTime, RationalTime.

9.1.1 Methods

absolute

```
public abstract AbsoluteTime176 absolute(Clock192 clock)
```

Convert the time of `this` to an absolute time, relative to the given instance of `Clock`192 . Convenient for situations where you really need an absolute time. Allocates a destination object if necessary. See the derived class comments for more specific information.

Parameters:

`clock` - An instance of `Clock`192 is used to convert the time of into absolute time.

absolute

```
public abstract AbsoluteTime176 absolute(Clock192 clock,
    AbsoluteTime176 dest)
```

Convert the time of `this` to an absolute time, relative to the given instance of `Clock`192 . Convenient for situations where you really need an absolute time. Allocates a destination object if necessary. See the derived class comments for more specific information.

Parameters:

`clock` - An instance of `Clock`192 is used to convert the time of into absolute time.

`dest` - If null, a new object is created and returned as result, else dest is returned.

compareTo

`public int compareTo(HighResolutionTime172 time)`

Compares this HighResolutionTime with the specified HighResolutionTime.

Parameters:

`time` - Compares with the time of `this`.

compareTo

`public int compareTo(java.lang.Object object)`

For the `Comparable` interface.

Specified By: `compareTo` in interface `Comparable`

equals

`public boolean equals(HighResolutionTime172 time)`

Returns true if the argument object has the same values as `this`.

Parameters:

`time` - Value compared to `this`.

equals

`public boolean equals(java.lang.Object object)`

Returns true if the argument is an instance of `HighResolutionTime` has the same values as this.

Overrides: `equals` in class `Object`

Parameters:

`object` - Value compared to `this`.

getMilliseconds

`public final long getMilliseconds()`

Returns the milliseconds component of `this`.

Returns: The milliseconds component of the time past the epoch represented by `this`.

getNanoseconds

`public final int **getNanoseconds**()`

Returns nanoseconds component of `this`.

hashCode

`public int **hashCode**()`

Overrides: `hashCode` in class `Object`

relative

`public abstract RelativeTime`$_{182}$ `**relative**(Clock`$_{192}$ `clock)`

Convert the time of `this` to a relative time, with respect to the given instance of `Clock`$_{192}$. Convenient for situations where you really need an relative time. Allocates a destination object if necessary. See the derived class comments for more specific information.

Parameters:

`clock` - An instance of `Clock`$_{192}$ is used to convert the time of into relative time.

relative

`public abstract RelativeTime`$_{182}$ `**relative**(Clock`$_{192}$ `clock, HighResolutionTime`$_{172}$ `dest)`

Convert the time of `this` to a relative time, with respect to the given instance of `Clock`$_{192}$. Convenient for situations where you really need an relative time. Allocates a destination object if necessary. See the derived class comments for more specific information.

Parameters:

`clock` - An instance of `Clock`$_{192}$ is used to convert the time of into relative time.

`dest` - If null, a new object is created and returned as result, else dest is returned.

set

`public void **set**(HighResolutionTime`$_{172}$ `time)`

~~Changes the time represented by the argument to some time between the invocation of the method and the return of the method.~~ Change the value represented by this to that of the given time. If the type of this and the type of the given time are not the same this method will throw IllegalArgument-Exception.

Parameters:

`time` - ~~The HighResolutionTime which will be set to represent the current time.~~ The new value for this.

set

`public void **set**(long millis)`

Sets the millisecond component of `this` to the given argument.

Parameters:

`millis` - This value will be the value of the millisecond component of this at the completion of the call. If `millis` is negative the millisecond value of this is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if a HighResolutionTime represnting time before the epoch is given as a parameter to the methods.

set

`public void **set**(long millis, int nanos)`

Sets the millisecond and nanosecond components of `this`.

Parameters:

`millis` - Value to set millisecond part of this. If `millis` is negative the millisecond value of this is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if a `HighResolutionTime` representing time before the epoch is given as a parameter to the methods.

`nanos` - Value to set nanosecond part of this. If `nanos` is negative the millisecond value of this is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if a `HighResolutionTime` representing time before the epoch is given as a parameter to the methods.

waitForObject

```
public static void waitForObject(java.lang.Object target,
    HighResolutionTime172 time)
    throws InterruptedException
```

Behaves exactly like `target.wait()` but with the enhancement that it waits with a precision of `HighResolutionTime`

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 this is `RelativeTime(0,0)` then wait indefinitely.

Throws:

`java.lang.InterruptedException` - If another threads interrupts this thread while its waiting.

See Also: `java.lang.Object.wait(long)`, `java.lang.Object.wait(long)`, `java.lang.Object.wait(long, int)`

9.2 AbsoluteTime

Declaration

`public class AbsoluteTime extends HighResolutionTime`$_{172}$

All Implemented Interfaces: `java.lang.Comparable`

Description

An object that represents a specific point in time given by milliseconds plus nanoseconds past the epoch (January 1, 1970, 00:00:00 GMT). This representation was designed to be compatible with the standard Java representation of an absolute time in the `java.util.Date` class.

If the value of any of the millisecond or nanosecond fields is negative the variable is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if an instance of `AbsoluteTime` representing time before the epoch is given as a parameter to the a method. For `add` and `substract` negative values behave just like 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

AbsoluteTime

`public AbsoluteTime()`

Equal to new `AbsoluteTime(0,0)`.

AbsoluteTime

`public AbsoluteTime(AbsoluteTime`$_{176}$ `time)`

Make a new `AbsolutTime` object from the given `AbsoluteTime` object.

Parameters:

`time` - The `AbsolutTime` object which is the source for the copy.

AbsoluteTime

`public AbsoluteTime(java.util.Date date)`

Equivalent to new `AbsoluteTime (date.getTime(),0)`.

Parameters:

`date` - The `java.util.Data` representation of the time past the epoch.

AbsoluteTime

`public AbsoluteTime(long millis, int nanos)`

Construct an `AbsoluteTime` object which means a time `millis` milliseconds plus `nanos` nanoseconds past 00:00:00 GMT on January 1, 1970.

Parameters:

`millis` - The milliseconds component of the time past the epoch.

`nanos` - The nanosecond component of the time past the epoch.

9.2.2 Methods

absolute

`public AbsoluteTime`$_{176}$ `absolute(Clock`$_{192}$ `clock)`

Convert time given by `this` to an absolute time relative to a given clock.

Overrides: `absolute`$_{172}$ in class `HighResolutionTime`$_{172}$

Parameters:

`clock` - `Clock`192 on which `this` will be based.

Returns: A reference to `this`.

absolute

`public AbsoluteTime`176 **`absolute`**`(Clock`192 `clock,`
`AbsoluteTime`176 `destination)`

Convert this time to an absolute time. For an `AbsoluteTime`, this is really easy: it just returns itself. Presumes that this time is already relative to the given clock.

Overrides: `absolute`172 in class `HighResolutionTime`172

Parameters:

`clock` - `Clock`192 on which this is based.

`destination` - Converted to an absolute time.

Returns: A reference to `this`.

add

`public AbsoluteTime`176 **`add`**`(long millis, int nanos)`

Add `millis` and `nanos` to `this`. A new object is allocated for the result.

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 `this` plus `millis` and `nanos`.

add

`public AbsoluteTime`176 **`add`**`(long millis, int nanos,`
`AbsoluteTime`176 `destination)`

Add `millis` and `nanos` to `this`. If the `destination` is non-null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Parameters:

`millis` - The number of milliseconds to be added to `this`.

`nanos` - The number of nanoseconds to be added to `this`.

`destination` - A reference to an `AbsoluteTime` object into which the result of the addition may be placed.

Returns: A reference to `destination` whose time is `this` plus `millis` and `nanos`.

add

`public final AbsoluteTime`$_{176}$ **`add`**`(RelativeTime`$_{182}$ `time)`

Add the time given by the parameter to `this`.

Parameters:

`time` - The time to add to `this`.

Returns: A reference to `this`.

add

`public AbsoluteTime`$_{176}$ **`add`**`(RelativeTime`$_{182}$ `time,`
`AbsoluteTime`$_{176}$ `destination)`

Add the time given by the parameter to `this`. If the `destination` is non-null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Parameters:

`time` - The time to add to `this`.

`destination` - A reference to an `AbsoluteTime` object into which the result of the addition may be placed.

Returns: A reference to `destination` or a new object whose time is `this` plus `millis` and `nanos`.

getDate

`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 miliseconds so the nanoseconds of `this` will be lost.

Returns: A reference to a `java.util.Date` object with a value of the time past the epoch represented by `this`.

relative

`public RelativeTime`$_{182}$ **`relative`**`(Clock`$_{192}$ `clock)`

Create a `RelativeTime`$_{182}$ object with the current time given by `this` with reference to the parameter.

Overrides: `relative`$_{174}$ in class `HighResolutionTime`$_{172}$

Parameters:

`clock` - The `Clock`192 reference used as a reference for `this`.

Returns: A reference to a new `RelativeTime`182 object whose time is set to the time given by `this` referencing the parameter.

relative

```
public RelativeTime182 relative(Clock192 clock,
    RelativeTime182 destination)
```

Create a relative time with the current time given by `this` with reference to the `Clock`192 parameter. If the `destination` is non-null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Parameters:

`clock` - The `Clock`192 reference used as a reference for `this`.

`destination` - A reference to a `RelativeTime`182 object into which the result of the subtraction may be placed.

Returns: A reference to a `RelativeTime`182 object whose time is set to the time given by `this` referencing the `Clock`192 parameter.

set

```
public void set(java.util.Date date)
```

Change the time represented by `this` to that given by the parameter.

Parameters:

`date` - A reference to a `java.util.Date` which will become the time represented by `this` after the completion of this method.

subtract

```
public final RelativeTime182 subtract(AbsoluteTime176 time)
```

Finds the difference between the time given by `this` and the time given by the parameter. The difference is, of course, a `RelativeTime`182 .

Parameters:

`time` - A reference to an `AbsoluteTime` object whose time is subtracted from `this`.

Returns: A reference to a new `RelativeTime`182 object whose time is the difference.

subtract

`public final RelativeTime`182 **`subtract`**`(AbsoluteTime`176 `time, RelativeTime`182 `destination)`

Finds the difference between the time given by `this` and the time given by the `AbsoluteTime` parameter. The difference is, of course, a `RelativeTime`182. If the `destination` is non-null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Parameters:

`time` - A reference to an `AbsoluteTime` object whose time is subtracted from `this`.

`destination` - A reference to an `RelativeTime`182 object into which the result of the addition may be placed.

Returns: A reference to a `RelativeTime`182 object whose time is the difference.

subtract

`public final AbsoluteTime`176 **`subtract`**`(RelativeTime`182 `time)`

Finds the absolute time that is the difference between `this` and the time given by the parameter.

Parameters:

`time` - A reference to a `RelativeTime`182 object whose time to subtract from `this`

Returns: A reference to a `AbsoluteTime` object whose time is the difference.

subtract

`public AbsoluteTime`176 **`subtract`**`(RelativeTime`182 `time, AbsoluteTime`176 `destination)`

Finds the absolute time that is the difference between the time given by `this` and the time given by the `RelativeTime`182 parameter. If the `destination` is non-null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Parameters:

`time` - A reference to an `RelativeTime`182 object whose time is subtracted from `this`.

`destination` - A reference to an `AbsoluteTime` object into which the result of the subtraction may be placed.

Returns: A reference to a `AbsoluteTime` object whose time is the difference between `this` and the `time` parameter.

toString

`public java.lang.String toString()`

Create a printable string of the time given by `this`, in a format that matches `java.util.Date.toString()` with a postfix to the detail the nanosecond value.

Overrides: `toString` in class `Object`

Returns: String object converted from the time given by `this`.

9.3 RelativeTime

Declaration

`public class RelativeTime extends HighResolutionTime`$_{172}$

All Implemented Interfaces: `java.lang.Comparable`

Direct Known Subclasses: `RationalTime`$_{187}$

Description

An object that represents a time interval milliseconds/10^3 + nanoseconds/10^9 seconds long. It generally is used to represent a time relative to now.

If the value of any of the millisecond or nanosecond fields is negative the variable is set to negative value. Although logically, and correctly, this may represent time before the epoch, invalid results *may* occur if an instance of `RelativeTime` representing time before the epoch is given as a parameter to some method. For `add` and `substract` negative values behave just like 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

RelativeTime

`public RelativeTime()`

Equivalent to `new RelativeTime(0,0)`.

RelativeTime

`public RelativeTime(long millis, int nanos)`

Construct a `RelativeTime` object representing an interval defined by the given parameter values.

Parameters:

`millis` - The milliseconds component.

`nanos` - The nanoseconds component.

RelativeTime

`public RelativeTime(RelativeTime`$_{182}$ `time)`

Make a new `RelativeTime` object from the given `RelativeTime` object

Parameters:

`time` - The `RelativeTime` object used as the source for the copy.

9.3.2 Methods

absolute

`public AbsoluteTime`$_{176}$ `absolute(Clock`$_{192}$ `clock)`

Convert the time given by `this`to an absolute time. The calculation is the current time indicated by the given instance of `Clock`$_{192}$ plus the interval given by `this`.

Overrides: `absolute`$_{172}$ in class `HighResolutionTime`$_{172}$

Parameters:

`clock` - The given instance of `Clock`$_{192}$. If null, `Clock.getRealTimeClock()` is used.

Returns: The new instance of `AbsoluteTime`$_{176}$ containing the result.

absolute

`public AbsoluteTime`$_{176}$ `absolute(Clock`$_{192}$ `clock, AbsoluteTime`$_{176}$ `destination)`

Convert the time given by `this`to an absolute time. The calculation is the current time indicated by the given instance of `Clock`$_{192}$ plus the interval given by `this`.

Overrides: `absolute`$_{172}$ in class `HighResolutionTime`$_{172}$

Parameters:

`clock` - The given instance of `Clock`$_{192}$. If null, `Clock.getRealTimeClock()` is used.

`destination` - A reference to the result instance of `AbsoluteTime`$_{176}$.

Returns: The new instance of `AbsoluteTime`$_{176}$ containing the result.

add

```
public RelativeTime182 add(long millis, int nanos)
```

Add a specific number of milliseconds and nanoseconds to the appropriate fields of `this`. A new object is allocated.

Parameters:

`millis` - The number of milliseconds to add.

`nanos` - The number of nanoseconds to add.

Returns: A new object containing the result of the addition.

add

```
public RelativeTime182 add(long millis, int nanos,
    RelativeTime182 destination)
```

Add a specific number of milliseconds and nanoseconds to the appropriate fields of `this`. A new object is allocated.

Parameters:

`millis` - The number of milliseconds to add.

`nanos` - The number of nanoseconds to add.

`destination` - A reference to the result instance of `RelativeTime`$_{182}$.

Returns: A new object containing the result of the addition.

add

```
public final RelativeTime182 add(RelativeTime182 time)
```

Add the interval of `this` to the interval of the given instance of `RelativeTime`$_{182}$.

Parameters:

`time` - A reference to the given instance of `RelativeTime`$_{182}$.

Returns: A new object containing the result of the addition.

add

`public RelativeTime`$_{182}$ **`add`**`(RelativeTime`$_{182}$ `time,`
`RelativeTime`$_{182}$ `destination)`

Add the interval of `this` to the interval of the given instance of `RelativeTime`$_{182}$.

Parameters:

`time` - A reference to the given instance of `RelativeTime`$_{182}$.

`destination` - A reference to the result instance of `RelativeTime`$_{182}$.

Returns: A new object containing the result of the addition.

addInterarrivalTo

`public void` **`addInterarrivalTo`**`(AbsoluteTime`$_{176}$ `timeAndDestination)`

Add the interval of `this` to the given instance of `AbsoluteTime`$_{176}$.

Parameters:

`timeAndDestination` - A reference to the given instance of `AbsoluteTime`$_{176}$ and the result.

getInterarrivalTime

`public RelativeTime`$_{182}$ **`getInterarrivalTime`**`()`

Gets the interval defined by `this`. For an instance of `RationalTime`$_{187}$ it is the interval divided by the frequency.

Returns: A reference to a new instance of `RelativeTime`$_{182}$ with the same interval as `this`.

getInterarrivalTime

`public RelativeTime`$_{182}$ **`getInterarrivalTime`**`(RelativeTime`$_{182}$
`destination)`

Gets the interval defined by `this`. For an instance of `RationalTime`$_{187}$ 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.

relative

`public RelativeTime`$_{182}$ **`relative`**`(Clock`$_{192}$ `clock)`

Make a new `RelativeTime` object from the time given by `this` but based on the given instance of `Clock`$_{192}$.

Overrides: `relative`$_{174}$ in class `HighResolutionTime`$_{172}$

Parameters:

`clock` - The given instance of `Clock`$_{192}$.

Returns: A reference to the new instance of `RelativeTime`$_{182}$.

relative

```
public RelativeTime182 relative(Clock192 clock,
     RelativeTime182 destination)
```

Make a new `RelativeTime` object from the time given by `this` but based on the given instance of `Clock`$_{192}$.

Parameters:

`clock` - The given instance of `Clock`$_{192}$.

`destination` - A reference to the result instance of `RelativeTime`$_{182}$.

Returns: A reference to the new instance of `RelativeTime`$_{182}$.

subtract

```
public final RelativeTime182 subtract(RelativeTime182 time)
```

Subtract the interval defined by the given instance of `RelativeTime`$_{182}$ from the interval defined by `this`.

Parameters:

`time` - A reference to the given instance of `RelativeTime`$_{182}$.

Returns: A new object holding the result of the subtraction.

subtract

```
public RelativeTime182 subtract(RelativeTime182 time,
     RelativeTime182 destination)
```

Subtract the interval defined by the given instance of `RelativeTime`$_{182}$ from the interval defined by `this`.

Parameters:

`time` - A reference to the given instance of `RelativeTime`$_{182}$.

`destination` - A reference to the object holding the result.

Returns: A new object holding the result of the subtraction.

toString

`public java.lang.String toString()`

Return a printable version of the time defined by `this`.

Overrides: `toString` in class `Object`

Returns: An instance of `java.lang.String` representing the time defined by `this`.

9.4 RationalTime

Declaration

`public class RationalTime extends RelativeTime`182

All Implemented Interfaces: `java.lang.Comparable`

Description

An object that represents a time interval milliseconds/10^3 + 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.

If the value of any of the millisecond or nanosecond fields is negative the variable is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if an instance of `AbsoluteTime`176 representing time before the epoch is given as a parameter to the a method.

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

RationalTime

`public RationalTime(int frequency)`

Construct an instance of `RationalTime`. Equivalent to new `RationalTime(1000, 0, frequency)`—essentially a cycles-per-second value.

RationalTime

`public` **`RationalTime`**`(int frequency, long millis, int nanos)`
`throws IllegalArgumentException`

Construct an instance of `RationalTime`. All arguments must be greater than or equal to zero.

Parameters:

`frequency` - The frequency value.

`millis` - The milliseonds value.

`nanos` - The nanoseconds value.

Throws:

`java.lang.IllegalArgumentException` - If any of the argument values are less than zero.

RationalTime

`public` **`RationalTime`**`(int frequency, RelativeTime`$_{182}$` interval)`
`throws IllegalArgumentException`

Construct an instance of `RationalTime` from the given `RelativeTime`$_{182}$.

Parameters:

`frequency` - The frequency value.

`interval` - The given instance of `RelativeTime`$_{182}$.

Throws:

`java.lang.IllegalArgumentException` - If any of the argument values are less than zero.

9.4.2 Methods

absolute

`public AbsoluteTime`$_{176}$ **`absolute`**`(Clock`$_{192}$` clock,`
`AbsoluteTime`$_{176}$` destination)`

Convert time of `this` to an absolute time.

Overrides: `absolute`$_{183}$ in class `RelativeTime`$_{182}$

Parameters:

`clock` - The reference clock. If null, `Clock.getRealTimeClock()` is used.

`destination` - A reference to the destination instance.

addInterarrivalTo

`public void addInterarrivalTo(AbsoluteTime176 destination)`

~~Add the time of thisto an~~ ~~AbsoluteTime176~~ It is almost the same dest.add(this, dest)~~ except that it accounts for (ie. divides by) the frequency.~~ This method should be deleted

Overrides: addInterarrivalTo185 in class RelativeTime182

Parameters:

`destination` - ~~A reference to the destination instance.~~

getFrequency

`public int getFrequency()`

Gets the value of `frequency`.

Returns: The value of `frequency` as an integer.

getInterarrivalTime

`public RelativeTime182 getInterarrivalTime()`

Gets the interarrival time. This time is (milliseconds/10^3 + nanoseconds/10^9)/frequency rounded down to the nearest expressible value of the fields and their types of RelativeTime182 .

Overrides: getInterarrivalTime185 in class RelativeTime182

getInterarrivalTime

`public RelativeTime182 getInterarrivalTime(RelativeTime182 dest)`

Gets the interarrival time. This time is (milliseconds/10^3 + nanoseconds/10^9)/frequency rounded down to the nearest expressible value of the fields and their types of RelativeTime182 .

Overrides: getInterarrivalTime185 in class RelativeTime182

Parameters:

`dest` - Result is stored in dest and returned, if null, a new object is returned.

set

```
public void set(long millis, int nanos)
    throws IllegalArgumentException
```

Sets the indicated fields to the given values.

Overrides: `set`$_{175}$ in class `HighResolutionTime`$_{172}$

Parameters:

`millis` - The new value for the millisecond field.

`nanos` - The new value for the nanosecond field.

Throws:

`java.lang.IllegalArgumentException`

setFrequency

```
public void setFrequency(int frequency)
    throws ArithmeticException
```

Sets the value of the `frequency` field.

Parameters:

`frequency` - The new value for the `frequency`.

Throws:

`java.lang.ArithmeticException`

CHAPTER 10 Timers

This section contains classes that:

- 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.

The classes provided by this section are `Clock`, `Timer`, `PeriodicTimer`, `and` `OneShotTimer`.

An instance of the `Clock` class is provided by the implementation. There is normally one clock provided, the system real-time clock. This object provides the mechanism for triggering behavior on expiration of a timer. It also reports the resolution of timers provided by the implementation.

An instance of `PeriodicTimer` fires an AsyncEvent at constant intervals.

An instance of `OneShotTimer` describes an event that is to be triggered exactly once at either an absolute time, or at a time relative to the creation of the timer. It may be used as the source for timeouts.

Instances of `Timer` are not used. The `Timer` class provides the interface and underlying implementation for both one-shot and periodic timers.

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 `Clock` class shall be capable of reporting the achievable resolution of timers

based on that clock.

2. The `OneShotTimer` class shall ensure that a one-shot timer is triggered exactly once, regardless of whether or not the timer is enabled after expiration of the indicated time.
3. The `PeriodicTimer` class shall allow the period of the timer to be expressed in terms of a `RelativeTime` or a `RationalTime`. In the latter case, the implementation shall provide a best effort to perform any correction necessary to maintain the frequency at which the event occurs.
4. If a periodic timer is enabled after expiration of the start time, the first event shall occur immediately and thus mark the start of the first period.

Rationale

The importance of the use of one-shot timers for timeout behavior and the vagaries in the execution of code prior to enabling the timer for short timeouts 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 for relative timers.

In many situations, it is important that a periodic task be represented as a frequency and that the period remain synchronized. In these cases, a relatively simple correction can be enforced by the implementation at the expense of some additional overhead for the timer.

10.1 Clock

Declaration

`public abstract class` **`Clock`**

Description

A clock advances from the past, through the present, into the future. 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. There are many possible subclasses of clocks: real-time clocks, user time clocks, simulation time clocks. The idea of using multiple clocks may at first seem unusual but we allow it as a possible resource allocation strategy. Consider a real-time system where the natural events of the system have different tolerances for jitter (jitter refers to the distribution of the differences between when the events are actually raised or noticed by the software and when they should have really occurred according to time in the real-world). Assume the system functions properly if event A is noticed or raised within plus or minus 100 seconds of the actual time it should occur but event B must be noticed or raised 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`197 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.

10.1.1 Constructors

Clock

`public Clock()`

Constructor for the abstract class.

10.1.2 Methods

getRealtimeClock

`public static Clock192 getRealtimeClock()`

There is always one clock object available: a realtime clock that advances in sync with the external world. This is the default `Clock`.

Returns: An instance of the default `Clock`

getResolution

`public abstract RelativeTime182 getResolution()`

Gets the resolution of the clock—the interval between ticks.

Returns: An instance of `RelativeTime`182 representing the resolution of `this`.

getTime

`public AbsoluteTime176 getTime()`

Gets the current time in a freshly allocated object.

Returns: An instance of `AbsoluteTime`176 representing the current time.

getTime

`public abstract void **getTime**(AbsoluteTime176 time)`

Gets the current time in an existing object. The time represented by the given `AbsoluteTime`176 is changed at some time between the invocation of the method and the return of the method.

Parameters:

`time` - The instance of `AbsoluteTime`176 object which will have its time changed. If null, then nothing happens.

setResolution

`public abstract void **setResolution**(RelativeTime182 resolution)`

Set the resolution of `this`. For some hardware clocks setting resolution impossible and if this method is called on those, then nothing happens.

Parameters:

`resolution` - The new resolution of `this`.

10.2 Timer

Declaration

`public abstract class **Timer** extends AsyncEvent207`

Direct Known Subclasses: `OneShotTimer`196, `PeriodicTimer`197

Description

A `Timer` is a timed event that measures time relative to a given `Clock`192 . This class defines basic functionality available to all timers. Applications will generally use either `PeriodicTimer`197 to create an event that is fired repeatedly at regular intervals, or `OneShotTimer`196 for an event that just fires once at a specific time. A timer is always based on a `Clock`192 , 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.

10.2.1 Constructors

Timer

```
protected Timer(HighResolutionTime172 time, Clock192 clock,
      AsyncEventHandler210 handler)
```

Create a timer that fires at the given time based on the given instance of `Clock`$_{192}$ and is handled by the specified handler.

Parameters:

`time` - The time to fire the event, Will be converted to absolute time.

`clock` - The clock on which to base this time. If null, the system realtime clock is used.

`handler` - The default handler to use for this event. If null, no handler is associated with it and nothing will happen when this event fires until a handler is provided.

10.2.2 Methods

createReleaseParameters

`public ReleaseParameters`$_{66}$ **`createReleaseParameters`**`()`

Create a `ReleaseParameters`$_{66}$ object appropriate to the release characteristics of this event. The default is the most pessimistic: `AperiodicParameters`$_{73}$. 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: `createReleaseParameters`$_{208}$ in class `AsyncEvent`$_{207}$

Returns: A new `ReleaseParameters`$_{66}$ object.

destroy

`public void` **`destroy`**`()`

Destroy thie timer and return all resources to the system.

disable

`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 continues to count while it is disabled and if it is subsequently reabled before its fire time occures and is enabled when its fire time occurs it will fire.

enable

`public void` **`enable`**`()`

Re-enable this timer after it has been disabled.

getClock

`public Clock`$_{192}$ **`getClock()`**

Gets the instance of `Clock`$_{192}$ that this timer is based on

Returns: clock The instance of `Clock`$_{192}$.

getFireTime

`public AbsoluteTime`$_{176}$ **`getFireTime()`**

Gets the time at which this event will fire.

Returns: An instance of `AbsoluteTime`$_{176}$ object representing the absolute time at which this will fire.

reschedule

`public void` **`reschedule`**`(HighResolutionTime`$_{172}$ `time)`

Change the scheduled time for this event. This method can take either absolute or relative times.

Parameters:

`time` - The time to reschedule for this event firing. If null, the previous fire time is still the time at which this will fire.

start

`public void` **`start()`**

Starts this time. A Timer starts measuring time from when it is started.

10.3 OneShotTimer

Declaration

`public class` **`OneShotTimer`** `extends Timer`$_{194}$

Description

A timed AsyncEvent that is driven by a clock. It will fire off once, when the clock time reaches the timeout time. If the clock time has already passed the timeout time, it will fire immediately.

10.3.1 Constructors

OneShotTimer

```
public OneShotTimer(HighResolutionTime172 time,
    AsyncEventHandler210 handler)
```

Create an instance of AsyncEvent that will excute its fire method at the expiration of the given time.

Parameters:

`time` - After timeout time units from 'now' fire will be excuted.

`handler` - The AsyncEventHandler that will be scheduled when fire is excuted.

OneShotTimer

```
public OneShotTimer(HighResolutionTime172 start, Clock192 clock,
    AsyncEventHandler210 handler)
```

Create an instance of AsyncEvent, based on the given clock, that will excute its fire method at the expiration of the given time.

Parameters:

`start` - Start time for timer.

`clock` - The timer will increment based on this clock.

`handler` - The AsyncEventHandler that will be scheduled when fire is excuted.

10.4 PeriodicTimer

Declaration

`public class PeriodicTimer extends Timer194`

Description

An `AsyncEvent207` whose fire method is executed periodically according to the given parameters. If a clock is given, calculation of the period uses the increments of the clock. If an interval is given or set the system guarantees that the fire method will execute interval time units after the last execution or its given start time as appropriate. If one of the `HighResolutionTime172` argument types is `RationalTime187` then the system guarantees that the fire method will be executed exactly frequency times every unit time (see `RationalTime187` constructors) by adjusting the interval between executions of `fire()`. This is similar to a thread with

PeriodicParameters 70 except that it is lighter weight. If a PeriodicTimer is disabled, it still counts, and if enabled at some later time, it will fire at its next scheduled fire time.

10.4.1 Constructors

PeriodicTimer

```
public PeriodicTimer(HighResolutionTime172 start,
      RelativeTime182 interval, AsyncEventHandler210 handler)
```

Create an instance of AsyncEvent that executes its fire method periodiacally

Parameters:

start - The time when the first interval begins

interval - The time between successive executions of the fire method

handler - The instance of AsyncEventHandler that will be scheduled each time the fire method is exceuted

PeriodicTimer

```
public PeriodicTimer(HighResolutionTime172 start,
      RelativeTime182 interval, Clock192 clock,
      AsyncEventHandler210 handler)
```

Create an instance of AsyncEvent that executes its fire method periodiacally

Parameters:

start - The time when the first interval begins

interval - The time between successive executions of the fire method

clock - The clock whose increments are used to calculate the interval

handler - The instance of AsyncEventHandler that will be scheduled each time the fire method is exceuted

10.4.2 Methods

createReleaseParameters

`public ReleaseParameters`$_{66}$ **`createReleaseParameters`**`()`

Create a `ReleaseParameters`$_{66}$ object with the next fire time as the start time and the interval of `this` as the period.

Overrides: `createReleaseParameters`$_{195}$ in class `Timer`$_{194}$

Returns: an instance of ReleaseParameters object

fire

`public void` **`fire`**`()`

This method will be private in 1.0.1

Overrides: `fire`$_{208}$ in class `AsyncEvent`$_{207}$

getFireTime

`public AbsoluteTime`$_{176}$ **`getFireTime`**`()`

Gets the absolute time `this` will next fire.

Overrides: `getFireTime`$_{196}$ in class `Timer`$_{194}$

Returns: The next fire time as an absolute time.

getInterval

`public void` **`getInterval`**`()`

Gets the interval of `this`.

Returns: The value in `interval`.

setInterval

`public void` **`setInterval`**`(RelativeTime`$_{182}$ `interval)`

Sets the `interval` value of `this`.

Parameters:

`interval` - The value to which `interval` will be set.

CHAPTER 11
Asynchrony

This section contains classes that:

- Provide mechanisms that bind the execution of program logic to the occurrence of internal and external events.
- Provide mechanisms that allow the asynchronous transfer of control.
- Provide mechanisms that allow the asynchronous termination of threads.

This specification provides several facilities for arranging asynchronous control of execution, some of which apply to threads in general while others apply only to real-time threads. These facilities fall into two main categories: asynchronous event handling and asynchronous transfer of control (ATC), which includes thread termination.

Asynchronous event handling is captured by the non-abstract class `AsyncEvent` and the abstract classes `AsyncEventHandler` and `BoundAsyncEventHandler`. An instance of the `AsyncEvent` class is an object corresponding to the possibility of an asynchronous event occurrence. An event occurrence may be initiated by either application logic or by the occurrence of a *happening* external to the JVM (such as a software signal or a hardware interrupt handler). An event occurrence is expressed in program logic by the invocation of the `fire()` method of an instance of the `AsyncEvent` class. The initiation of an event occurrence due to a happening is implementation dependent.

An instance of the class `AsyncEventHandler` is an object embodying code that is scheduled in response to the occurrence of an event. The `run()` method of an instance of `AsyncEventHandler` acts like a thread, and indeed one of its constructors takes references to instances of `SchedulingParameters`, `ReleaseParameters`, and `MemoryParameters`. However, there is not necessarily a separate thread for each `run()` method. The class `BoundAsyncEventHandler` extends `AsyncEventHandler,` and should be used if it is necessary to ensure that a handler has a dedicated thread.

An event count is maintained so that a handler can cope with event bursts —- situations where an event is fired 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()` and adds an overloaded 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` (AIE), its subclass `Timed`, the interface `Interruptible`, and in the semantics of the interrupt methods in `Thread` and `RealtimeThread`.

A method indicates its willingness to be asynchronously interrupted by including AIE on its `throws` clause. If a thread is asynchronously interrupted while executing a method that identifies AIE on its `throws` clause, then an instance of AIE will be thrown as soon as the thread 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

This list establishes the semantics and requirements that are applicable to `AsyncEvent` objects. 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 instance of `AsyncEvent` occurs (by either program logic or a happening), all `run()` methods of instances of the `AsyncEventHandler` class that have been added to the instance of `AsyncEvent` by the execution of `addHandler()` are scheduled for execution. This action may or may not be idempotent. Every occurrence of an event increments a counter in each associated handler. Handlers may elect to execute logic for each occurrence of the event or not.
2. Instances of `AsyncEvent` and `AsyncEventHandler` may be created and used by any program logic.
3. More than one instance of `AsyncEventHandler` may be added to an instance of `AsyncEvent`.
4. An instance of `AsyncEventHandler` may be added to more than one instance of

AsyncEvent.

This list establishes the semantics and requirements that are applicable to AsynchronouslyInterruptedException. 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. Instances of the class AsynchronouslyInterruptedException can be generated by execution of program logic and by internal virtual machine mechanisms that are asynchronous to the execution of program logic which is the target of the exception.
2. Program logic that exists in methods that throw AsynchronouslyInterruptedException is subject to receiving an instance of AsynchronouslyInterruptedException at any time during execution except as provided below.
3. The RTSJ specifically requires that blocking methods in java.io.* must be prevented from blocking indefinitely when invoked from a method with AIE in its throws clause. The implementation, when either AIE.fire() or RealtimeThread.interrupt() is called when control is in a java.io.* method invoked from an interruptible method, may either unblock the blocked call, raise an IOException on behalf of the call, or allow the call to complete normally if the implementation determines that the call would eventually unblock.
4. If control is in the lexical scope of an ATC-deferred section when an AIE is raised by fire() on an AIE object or interrupt() on the current real-time thread object, the AIE is made pending. The exception does not initiate the transfer of control corresponding to the AIE until the first subsequent attempt to transfer control to code that is not ATC-deferred. At that point control is transferred to the nearest dynamically-enclosing catch clause of a try statement that handles this AIE and which is in an ATC-deferred section. 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 JLS definitions unaltered.
5. ~~Program logic executing within a synchronized block within a method with AsynchronouslyInterruptedException in its throws clause is not subject to receiving an instance of AIE. The interrupted state of the execution context is set to pending and the program logic will receive the instance when control passes out of the synchronized block if other semantics in this list so indicate.~~
6. Constructors are allowed to include AsynchronouslyInterruptedException in their throws clause and will thus be interruptible.
7. A thread that is subject to asynchronous interruption (in a method that throws AIE, but not in a synchronized block) must respond to that exception within a bounded number of bytecodes. This worst-case response interval (in bytecode instructions) must be documented.

Definitions
The RTSJ's approach to ATC is designed to follow these principles. It is based on exceptions and is an extension of the current Java language rules for `java.lang.Thread.interrupt()`. The following terms and abbreviations will be used:

ATC - Asynchronous Transfer of Control

AIE - (Asynchronously Interrupted Exception) The class `javax.realtime.AsynchronouslyInterruptedException`, a subclass of `java.lang.InterruptedException`.

AI-method - (Asynchronously Interruptible) A method is said to be asynchronously interruptible if it includes AIE in its throws clause.

ATC-deferred section - a synchronized method, a synchronized statement, or any method or constructor without AIE in its throws clause.

Summary of Operation
In summary, ATC works as follows:

If `t` is an instance of `RealtimeThread` or `NoHeapRealtimeThread` and `t.interrupt()` or `AIE.fire()` is executed by any thread in the system then:

1. If control is in an ATC-deferred section, then the AIE is put into a pending state.
2. If control is not in an ATC-deferred section, then control is transferred to the nearest dynamically-enclosing `catch` clause of a `try` statement that handles this AIE and which **is in** an ATC-deferred section. 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 either `wait()`, `sleep()`, or `join()`, the thread is awakened and the fired AIE (which is a subclass of `InterruptedException)` is thrown. Then ATC follows option 1, or 2 as appropriate.
4. If control is in a non-AI method, control continues normally until the first attempt to return to an AI method or invoke an AI method. Then ATC follows option 1, or 2 as appropriate.
5. If control is transferred from a non-AI method 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.

If an AIE is in a pending state then this AIE is thrown only when:

1. Control enters an AI-method.
2. Control returns to an AI-method.
3. Control leaves a synchronized block within an AI-method.

When happened() is called on an AIE or that AIE is superseded by another the first AIE's state is made non-pending.

	Match	No Match
Propagate == true	clear the pending AIE, return true	propagate (whether the AIE remains pending is invisible except to the implementation)
Propagate == false	clear the pending AIE, return false	do not clear the pending AIE, return false

An AIE may be raised while another AIE is pending or in action. 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 t.interrupt() is invoked and AIE_i (i = 1,...,n, for n unique instances of AIE) be the AIE raised when AIE_i.fire() is invoked. Assume stacks grow down and therefore the phrase "a frame lower on the stack than this frame" refers to a method at a deeper nesting level.

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 lower 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 higher on the stack, the current AIE (AIE_x) is replaced by the new AIE (AIE_y).

Rationale

The design of the asynchronous event handling was intended to provide the necessary functionality while allowing efficient implementations and catering to 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 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 thread (the class AsyncEventHandler) or alternatively as bound to a thread (BoundAsyncEventHandler).

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 for one thread to trigger an ATC in another thread 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`, or AIE), the way that a method indicates its susceptibility to ATC is by including AIE on its `throws` clause. Causing this exception to be thrown in a 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.

Constructors and `finally` clauses are subject to interruption. If a constructor is aborted, an object might be only partially initialized. If a `finally` clause is aborted, needed cleanup code might not be performed. It is the programmer's responsibility to ensure that executing these constructs does not induce unwanted ATC latency. 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()` prone to deadlock.

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 instance of 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 thread is asynchronously interrupted while in a try block that has a handler such as

```
catch (Throwable e){ return; }
```

then the AIE instance will still be propagated to the caller.

This specification does not provide a special mechanism for terminating a thread; ATC can be used to achieve this effect. This means that, by default, a thread cannot be terminated; it needs to invoke methods that have AIE in their `throws` clauses. 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`$_{194}$

Description

An asynchronous event represents something that can happen, like a light turning red. It can have a set of handlers associated with it, and when the event occurs, the handler is scheduled by the scheduler to which it holds a reference (see `AsyncEventHandler`$_{210}$ and `Scheduler`$_{54}$).

A major motivator for this style of building events is that we expect to have lots of events and lots of event handlers. An event handler is logically very similar to a thread, but it is intended to have a much lower cost (in both time and space)—assuming that a relatively small number of events are fired and in the process of being handled at once. `AsyncEvent.fire()` differs from a method call because the handler (a) has scheduling parameters and (b) is executed asynchronously.

11.1.1 Constructors

AsyncEvent

`public` **`AsyncEvent`**`()`

Create a new `AsyncEvent` object.

11.1.2 Methods

addHandler

`public void` **`addHandler`**`(AsyncEventHandler`$_{210}$ `handler)`

Add a handler to the set of handlers associated with this event. An instance of `AsyncEvent` may have more than one associated handler.

Parameters:

`handler` - The new handler to add to the list of handlers already associated with this. If `handler` is null then nothing happens.

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

bindTo

```
public void bindTo(java.lang.String happening)
    throws UnknownHappeningException
```

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 occurs.

Parameters:

`happening` - An implementation dependent value that binds this instance of `AsyncEvent` to a happening.

Throws:

`UnknownHappeningException`$_{256}$ - If the String value is not supported by the implementation.

createReleaseParameters

```
public ReleaseParameters66 createReleaseParameters()
```

Create a `ReleaseParameters`$_{66}$ object appropriate to the release characteristics of this event. The default is the most pessimistic: `AperiodicParameters`$_{73}$. 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.

Returns: A new `ReleaseParameters`$_{66}$ object.

fire

```
public void fire()
```

Fire this instance of `AsyncEvent`. The `run()` methods of instances of `AsyncEventHandler`$_{210}$ associated with this event will be made ready to run.

handledBy

```
public boolean handledBy(AsyncEventHandler210 handler)
```

Returns true if and only 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 `target` is null of the parameters is not associated with `this`.

removeHandler

`public void` **`removeHandler`**`(AsyncEventHandler`$_{210}$ `handler)`

Remove a handler from the set associated with this event.

Parameters:

`handler` - The handler to be disassociated from `this`. If null nothing happens. If not already associated with this then nothing happens.

setHandler

`public void` **`setHandler`**`(AsyncEventHandler`$_{210}$ `handler)`

Associate a new handler with this event and remove all existing handlers.

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

Parameters:

`handler` - The new instance of `AsyncEventHandler`$_{210}$ to be associated with this. If `handler` is null then no handler will be associated with this (i.e., remove all handlers).

unbindTo

```
public void unbindTo(java.lang.String happening)
     throws UnknownHappeningException
```

Removes a binding to an external event, a *happening*. The meaningful values of `happening` are implementation dependent.

Parameters:

`happening` - An implementation dependent value representing some external event to which this instance of `AsyncEvent` is bound.

Throws:

`UnknownHappeningException`$_{256}$ - If this instance of `AsyncEvent` is not bound to the given `happening` or the given `java.lang.String` value is not supported by the implementation.

11.2 AsyncEventHandler

Declaration

`public class` **`AsyncEventHandler`** `implements Schedulable`$_{47}$

All Implemented Interfaces: `java.lang.Runnable`, `Schedulable`$_{47}$

Direct Known Subclasses: `BoundAsyncEventHandler`$_{224}$

Description

An asynchronous event handler encapsulates code that executes at some time after an instance of `AsyncEvent`$_{207}$ to which it is bound occurs.

It is essentially a `java.lang.Runnable` with a set of parameter objects, making it very much like a `RealtimeThread`$_{25}$. The expectation is that there may be thousands of events, with corresponding handlers, averaging about one handler per event. The number of unblocked (i.e., scheduled) handlers is expected to be relatively small.

It is guaranteed that multiple firings of an event handler will be serialized. It is also guaranteed that (unless the handler explicitly chooses otherwise) for each firing of the handler, there will be one execution of the `handleAsyncEvent()`$_{218}$ method.

Instances of `AsyncEventHandler` with a release parameter of type `SporadicParameters`$_{75}$ have a list of release times which correspond to the occurance times of instances of `AsyncEvent`$_{207}$ to which it is bound. The minimum interarrival time specified in `SporadicParameters`$_{75}$ is enforced as defined there. Unless the handler explicitly chooses otherwise there will be one execution of the code in `handleAsyncEvent()`$_{218}$ for each entry in the list. The ith execution of `handleAsyncEvent()`$_{218}$ will be released for scheduling at the time of the ith entry in the list.

There is no restriction on what handlers may do. They may run for a long or short time, and they may block. (Note: blocked handlers may hold system resources.)

Normally, handlers are bound to an execution context dynamically, when the instance of `AsyncEvent`$_{207}$ to which they are bound occurs. This can introduce a (small) time penalty. For critical handlers that can not afford the expense, and where this penalty is a problem, use a `BoundAsyncEventHandler`$_{224}$.

The semantics for memory areas that were defined for realtime 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 realtime threads.

11.2.1 Constructors

AsyncEventHandler

`public **AsyncEventHandler**()`

Create an instance of `AsyncEventHandler` whose `SchedulingParameters`63 are inherited from the current thread and does not have either `ReleaseParameters`66 or `MemoryParameters`150 .

AsyncEventHandler

`public **AsyncEventHandler**(boolean nonheap)`

Create an instance of `AsyncEventHandler` whose parameters are inherited from the current thread, if the current thread is a `RealtimeThread`25 , or null, otherwise.

Parameters:

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`38 . A false value means this will have characteristics identical to a `RealtimeThread`25 . If true and the current thread is *not* a `NoHeapRealtimeThread`38 or a `RealtimeThread`25 executing within a `ScopedMemory`98 or `ImmortalMemory`96 scope then an IllegalArgumentException is thrown.

Throws:

`java.lang.IllegalArgumentException` - If the initial memory area is in heap memory, and the `noheap` parameter is true.

AsyncEventHandler

`public **AsyncEventHandler**(java.lang.Runnable logic)`

Create an instance of `AsyncEventHandler` whose `SchedulingParameters`63 are inherited from the current thread and does not have either `ReleaseParameters`66 or `MemoryParameters`150 .

Parameters:

`logic` - The `java.lang.Runnable` object whose `run()` method is executed by `handleAsyncEvent()`218 .

AsyncEventHandler

```
public AsyncEventHandler(java.lang.Runnable logic,
        boolean nonheap)
```

Create an instance of `AsyncEventHandler` whose parameters are inherited from the current thread, if it is a `RealtimeThread`25 , or null otherwise.

Parameters:

`logic` - The `java.lang.Runnable` object whose `run()` method is executed by `handleAsyncEvent()`218 .

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`38 . A false value means this will have characteristics identical to a `RealtimeThread`25 . If true and the current thread is *not* a `NoHeapRealtimeThread`38 or a `RealtimeThread`25 executing within a `ScopedMemory`98 or `ImmortalMemory`96 scope then an IllegalArgumentException is thrown.

Throws:

`java.lang.IllegalArgumentException` - If the initial memory area is in heap memory, and the `noheap` parameter is true.

AsyncEventHandler

```
public AsyncEventHandler(SchedulingParameters63 scheduling,
        ReleaseParameters66 release, MemoryParameters150 memory,
        MemoryArea90 area, ProcessingGroupParameters81 group,
        boolean nonheap)
```

Create an instance of `AsyncEventHandler` with the specifed parameters.

Parameters:

`scheduling` - A `SchedulingParameters`63 object which will be associated with the constructed instance. If null, `this` will be assigned the reference to the `SchedulingParameters`63 of the current thread.

`release` - A `ReleaseParameters`66 object which will be associated with the constructed instance. If null, `this` will have no `ReleaseParameters`66 .

`memory` - A `MemoryParameters`150 object which will be associated with the constructed instance. If null, `this` will have no `MemoryParameters`150 .

`area` - The `MemoryArea`90 for `this`. If null, the memory area will be that of the current thread.

`group` - A `ProcessingGroupParameters`81 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 `NoHeapRealtimeThread`38 . A false value means this will have characteristics identical to a `RealtimeThread`25 . If true and the current thread is *not* a `NoHeapRealtimeThread`38 or a `RealtimeThread`25 executing within a `ScopedMemory`98 or `ImmortalMemory`96 scope then an IllegalArgumentException is thrown.

Throws:

`java.lang.IllegalArgumentException` - If the initial memory area is in heap memory, and the `noheap` parameter is true.

AsyncEventHandler

```
public AsyncEventHandler(SchedulingParameters63 scheduling,
    ReleaseParameters66 release, MemoryParameters150 memory,
    MemoryArea90 area, ProcessingGroupParameters81 group,
    java.lang.Runnable logic)
```

Create an instance of `AsyncEventHandler` with the specified parameters.

Parameters:

`scheduling` - A `SchedulingParameters`63 object which will be associated with the constructed instance. If null, `this` will be assigned the reference to the `SchedulingParameters`63 of the current thread.

`release` - A `ReleaseParameters`66 object which will be associated with the constructed instance. If null, `this` will have no `ReleaseParameters`66 .

`memory` - A `MemoryParameters`150 object which will be associated with the constructed instance. If null, `this` will have no `MemoryParameters`150 .

`area` - The `MemoryArea`90 for `this`. If null, the memory area will be that of the current thread.

`group` - A `ProcessingGroupParameters`81 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 `handleAsyncEvent()`218 .

AsyncEventHandler

```
public AsyncEventHandler(SchedulingParameters63 scheduling,
     ReleaseParameters66 release, MemoryParameters150 memory,
     MemoryArea90 area, ProcessingGroupParameters81 group,
     java.lang.Runnable logic, boolean nonheap)
```

Create an instance of `AsyncEventHandler` with the specifed parameters.

Parameters:

`scheduling` - A `SchedulingParameters`$_{63}$ object which will be associated with the constructed instance. If null, `this` will be assigned the reference to the `SchedulingParameters`$_{63}$ of the current thread.

`release` - A `ReleaseParameters`$_{66}$ object which will be associated with the constructed instance. If null, `this` will have no `ReleaseParameters`$_{66}$.

`memory` - A `MemoryParameters`$_{150}$ object which will be associated with the constructed instance. If null, `this` will have no `MemoryParameters`$_{150}$.

`area` - The `MemoryArea`$_{90}$ for `this`. If null, the memory area will be that of the current thread.

`group` - A `ProcessingGroupParameters`$_{81}$ 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 `handleAsyncEvent()`$_{218}$.

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`$_{38}$. A false value means this will have characteristics identical to a `RealtimeThread`$_{25}$. If true and the current thread is *not* a `NoHeapRealtimeThread`$_{38}$ or a `RealtimeThread`$_{25}$ executing within a `ScopedMemory`$_{98}$ or `ImmortalMemory`$_{96}$ scope then an IllegalArgumentException is thrown.

Throws:

`java.lang.IllegalArgumentException` - If the initial memory area is in heap memory, and the `noheap` parameter is true.

11.2.2 Methods

addIfFeasible

`public boolean addIfFeasible()`

Inform the scheduler and cooperating facilities that the feasibility parameters assiciated with `this` should be considered in feasibility analyses until further notified, only if the new set of parameters is feasible.

Specified By: `addIfFeasible`47 in interface `Schedulable`47

Returns: True if the addition is successful. False if the addition is not successrul or there is no assigned scheduler.

addToFeasibility

`public boolean addToFeasibility()`

Inform the scheduler and cooperating facilities that the feasibility parameters assiciated with `this` should be considered in feasibility analyses until further notified.

Specified By: `addToFeasibility`48 in interface `Schedulable`47

getAndClearPendingFireCount

`protected final 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 used by handlers for which the logic can accommodate multiple firings in a single execution. The general form for using this is:

```
public void handleAsyncEvent() {
    int numberOfFirings = getAndClearPendingFireCount();
    <handle the events>
}
```

Returns: The value held by `fireCount` prior to setting the value to zero.

getAndDecrementPendingFireCount

`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 firings:

```
public void handleAsyncEvent() {
   <setup>
   do {
      <handle the event>
      } while(getAndDecrementPendingFireCount()>0);
}
```

This construction is necessary only in the case where one wishes to avoid the setup costs since the framework guarantees that `handleAsyncEvent()` will be invoked the appropriate number of times.

Returns: The value held by `fireCount` prior to decrementing it by one.

getAndIncrementPendingFireCount

`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.

Returns: The value held by `fireCount` prior to incrementing it by one.

getMemoryArea

`public MemoryArea`$_{90}$ **`getMemoryArea`**`()`

This is an accessor method for the instance of `MemoryArea`$_{90}$ associated with `this`.

Returns: The instance of `MemoryArea`$_{90}$ which is the current area for `this`.

getMemoryParameters

`public MemoryParameters`$_{150}$ **`getMemoryParameters`**`()`

Gets the memory parameters associated with this instance of `Schedulable`$_{47}$.

Specified By: `getMemoryParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: The `MemoryParameters`$_{150}$ object associated with `this`.

getPendingFireCount

`protected final int` **`getPendingFireCount`**`()`

This is an accessor method for `fireCount`. The `fireCount` field nominally holds the number of times associated instances of `AsyncEvent`$_{207}$ have occurred that have not had the method `handleAsyncEvent()` invoked. Due to accessor methods the application logic may manipulate the value in this field for application specific reasons.

Returns: The value held by `fireCount`.

getProcessingGroupParameters

`public ProcessingGroupParameters`$_{81}$ **`getProcessingGroupParameters`**`()`

Gets the processing group parameters associated with this instance of `Schedulable`$_{47}$.

Specified By: `getProcessingGroupParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: The `ProcessingGroupParameters`$_{81}$ object assiciated with `this`.

getReleaseParameters

`public ReleaseParameters`$_{66}$ **`getReleaseParameters`**`()`

Gets the release parameters associated with this instance of `Schedulable`$_{47}$.

Specified By: `getReleaseParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: The `ReleaseParameters`$_{66}$ object associated with `this`.

getScheduler

`public Scheduler`$_{54}$ **`getScheduler`**`()`

Gets the instance of `Scheduler`$_{54}$ associated with this instance of `Schedulable`$_{47}$.

Specified By: `getScheduler`$_{48}$ in interface `Schedulable`$_{47}$

Returns: The instance of `Scheduler`$_{54}$ associated with `this`.

getSchedulingParameters

`public SchedulingParameters`$_{63}$ **`getSchedulingParameters`**`()`

Gets the scheduling parameters associated with this instance of `Schedulable`$_{47}$.

Specified By: `getSchedulingParameters`$_{48}$ in interface `Schedulable`$_{47}$

Returns: The `SchedulingParameters`$_{63}$ object associated with `this`.

handleAsyncEvent

`public void handleAsyncEvent()`

This method holds the logic which is to be executed when assiciated instances of `AsyncEvent`$_{207}$ occur. If this handler was constructed using an instance of `java.lang.Runnable` as an argument to the constructor, then that instance's `run()` method will be inviked from this method. This method will be invoked repeatedly while `fireCount` is greater than zero.

removeFromFeasibility

`public boolean removeFromFeasibility()`

Inform the scheduler and cooperating facilities that the scheduling characteristics of this instance of `Schedulable`$_{47}$ should not be considered in feasibility analyses until further notified.

Specified By: `removeFromFeasibility`$_{49}$ in interface `Schedulable`$_{47}$

Returns: True, if the removal was successful. False, if the removal was unsuccessful.

run

`public final void run()`

Used by the asynchronous event mechanism, see `AsyncEvent`$_{207}$. This method invokes `handleAsyncEvent()` repeatedly while the fire count is greater than zero. Applications cannot override this method and should thus override `handleAsyncEvent()` in subclasses with the logic of the handler.

Specified By: `run` in interface `Runnable`

setIfFeasible

`public boolean setIfFeasible(ReleaseParameters`$_{66}$ `release, MemoryParameters`$_{150}$ `memory)`

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible`49 in interface `Schedulable`47

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

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.

setIfFeasible

```
public boolean setIfFeasible(ReleaseParameters66 release,
      MemoryParameters150 memory,
      ProcessingGroupParameters81 group)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`47 or an instance of `Schedulable`47. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`47. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`47 as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible`49 in interface `Schedulable`47

Parameters:

`release` - The proposed release parameters.

`memory` - The proposed memory parameters.

`group` - The proposed processing group parameters.

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.

setIfFeasible

```
public boolean setIfFeasible(ReleaseParameters66 release,
      ProcessingGroupParameters81 group)
```

This method appears in many classes in the RTSJ and with various parameters. The parameters are either new scheduling characteristics for an instance `Schedulable`$_{47}$ or an instance of `Schedulable`$_{47}$. The method first performs a feasibility analysis using the new scheduling characteristics as replacements for the matching scheduling characteristics of either `this` or the given instance of `Schedulable`$_{47}$. If the resulting system is feasible the method replaces the current scheduling characteristics, of either `this` or the given instance of `Schedulable`$_{47}$ as appropriate, with the new scheduling characteristics.

Specified By: `setIfFeasible`$_{50}$ in interface `Schedulable`$_{47}$

Parameters:

`release` - The proposed release parameters.

`group` - The proposed processing group parameters.

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.

setMemoryParameters

`public void` **`setMemoryParameters`**`(MemoryParameters`$_{150}$ `memory)`

Sets the memory parameters associated with this instance of `Schedulable`$_{47}$. When it is next executed, that execution will use the new parameters to control memory allocation. Does not affect the current invocation of the `run()` of this handler.

Specified By: `setMemoryParameters`$_{50}$ in interface `Schedulable`$_{47}$

Parameters:

`memory` - A `MemoryParameters`$_{150}$ object which will become the memory parameters associated with this after the method call.

setMemoryParametersIfFeasible

`public boolean` **`setMemoryParametersIfFeasible`**`(MemoryParameters`$_{150}$ `memory)`

The method first performs a feasibility analysis using the given memory parameters as replacements for the memory parameters of `this` If the resulting system is feasible the method replaces the current memory parameters of `this` with the new memory parameters.

Specified By: `setMemoryParametersIfFeasible`$_{51}$ in interface `Schedulable`$_{47}$

Parameters:

`memory` - The proposed memory parameters.

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.

setProcessingGroupParameters

```
public void
    setProcessingGroupParameters(ProcessingGroupParameters81
    group)
```

Sets the processing group parameters associated with this instance of `Schedulable`47.

Specified By: `setProcessingGroupParameters`51 in interface `Schedulable`47

Parameters:

`parameters` - A `ProcessingGroupParameters`81 object which will become the processing group parameters associated with `this` after the method call.

setProcessingGroupParametersIfFeasible

```
public boolean
    setProcessingGroupParametersIfFeasible(ProcessingGroupParam
    eters81 group)
```

Need description.

Specified By: `setProcessingGroupParametersIfFeasible`51 in interface `Schedulable`47

setReleaseParameters

```
public void setReleaseParameters(ReleaseParameters66 release)
```

Sets the release parameters associated with this instance of `Schedulable`47. When it is next executed, that execution will use the new parameters to control scheduling. If the scheduling parameters of a handler is set to null, the handler will be executed immediately when any associated `AsyncEvent`207 is fired, in the context of the thread invoking the `fire()` method. Does not affect the current invocation of the `run()` of this handler.

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

Specified By: `setReleaseParameters`52 in interface `Schedulable`47

Parameters:

`release` - A `ReleaseParameters`66 object which will become the release parameters associated with this after the method call.

setReleaseParametersIfFeasible

```
public boolean setReleaseParametersIfFeasible(ReleaseParameters66
      release)
```

Need description.

Specified By: `setReleaseParametersIfFeasible`52 in interface `Schedulable`47

setScheduler

```
public void setScheduler(Scheduler54 scheduler)
      throws IllegalThreadStateException
```

Sets the scheduler associated with this instance of `Schedulable`47.

Specified By: `setScheduler`52 in interface `Schedulable`47

Parameters:

`scheduler` - An instance of `Scheduler`54 which will manage the execution of this thread. If `scheduler` is null nothing happens.

Throws:

`java.lang.IllegalThreadStateException`

setScheduler

```
public void setScheduler(Scheduler54 scheduler,
      SchedulingParameters63 scheduling,
      ReleaseParameters66 release, MemoryParameters150 memory,
      ProcessingGroupParameters81 group)
      throws IllegalThreadStateException
```

Sets the scheduler associated with this instance of `Schedulable`47.

Specified By: `setScheduler`53 in interface `Schedulable`47

Parameters:

`scheduler` - An instance of `Scheduler`54 which will manage the execution of this thread. If `scheduler` is null nothing happens.

`scheduling` - A `SchedulingParameters`63 object which will be associated with `this`. If null, `this` will be assigned the reference to the instance of `SchedulingParameters`63 of the current thread.

`release` - A `ReleaseParameters`66 object which will be associated with `this`. If null, `this` will have no associated instance of `ReleaseParameters`66 .

`memory` - A `MemoryParameters`150 object which will be associated with `this`. If null, `this` will have no associated instance of `MemoryParameters`150 .

`group` - A `ProcessingGroupParameters`81 object which will be associated with `this`. If null, `this` will have no associated instance of `ProcessingGroupParameters`81 .

Throws:

`java.lang.IllegalThreadStateException`

setSchedulingParameters

```
public void setSchedulingParameters(SchedulingParameters63
    scheduling)
```

Sets the scheduling parameters associated with this instance of `Schedulable`47 . When it is next executed, that execution will use the new parameters to control releases. If the scheduling parameters of a handler is set to null, the handler will be executed immediately when any associated `AsyncEvent`207 is fired, in the context of the thread invoking the `fire()` method. Does not affect the current invocation of the `run()` of this handler.

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

Specified By: `setSchedulingParameters`53 in interface `Schedulable`47

Parameters:

`parameters` - A `SchedulingParameters`63 object which will become the scheduling parameters associated with `this` after the method call.

setSchedulingParametersIfFeasible

```
public boolean
    setSchedulingParametersIfFeasible(SchedulingParameters63
    scheduling)
```

The method first performs a feasibility analysis using the given scheduling parameters as replacements for the scheduling parameters of `this` If the resulting system is feasible the method replaces the current scheduling parameters of `this` with the new scheduling parameters.

Specified By: `setSchedulingParametersIfFeasible`54 in interface `Schedulable`47

Parameters:

`scheduling` - The proposed scheduling parameters.

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.

11.3 BoundAsyncEventHandler

Declaration

```
public abstract class BoundAsyncEventHandler extends
             AsyncEventHandler210
```

All Implemented Interfaces: `java.lang.Runnable`, `Schedulable`47

Description

A bound asynchronous event handler is an instance of `AsyncEventHandler`210 that is permanently bound to a thread. Bound asynchronous event handlers are meant for use in situations where the added timeliness is worth the overhead of binding the handler to a thread.

11.3.1 Constructors

BoundAsyncEventHandler

```
public BoundAsyncEventHandler()
```

Create a handler whose parameters are inherited from the current thread, if it is a `RealtimeThread`25, or null otherwise.

BoundAsyncEventHandler

```
public BoundAsyncEventHandler(SchedulingParameters63 scheduling,
     ReleaseParameters66 release, MemoryParameters150 memory,
     MemoryArea90 area, ProcessingGroupParameters81 group,
     java.lang.Runnable logic, boolean nonheap)
```

Create a handler with the specified parameters.

Parameters:

`scheduling` - A `SchedulingParameters`63 object which will be associated with the constructed instance. If null, `this` will be assigned the reference to the `SchedulingParameters`63 of the current thread.

`release` - A `ReleaseParameters`66 object which will be associated with the constructed instance. If null, `this` will have no `ReleaseParameters`66 .

`memory` - A `MemoryParameters`150 object which will be associated with the constructed instance. If null, `this` will have no `MemoryParameters`150 .

`area` - The `MemoryArea`90 for `this`. If null, the memory area will be that of the current thread.

`group` - A `ProcessingGroupParameters`81 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()`218 .

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`38 . A false value means this will have characteristics identical to a `RealtimeThread`25 . If true and the current thread is *not* a `NoHeapRealtimeThread`38 or a `RealtimeThread`25 executing within a `ScopedMemory`98 or `ImmortalMemory`96 scope then an IllegalArgumentException is thrown.

Throws:

`java.lang.IllegalArgumentException` - If the initial memory area is in heap memory, and the `noheap` parameter is true.

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 `AsynchronouslyInterruptedException`226 and its subclasses. `doInterruptible()` invokes the implementation of the method in this interface. Thus the *system* can ensure correctness before invoking `run()` and correctly cleaning up after `run()` returns.

11.4.1 Methods

interruptAction

`public void` **`interruptAction`**`(AsynchronouslyInterruptedException`$_{226}$ `exception)`

This method is called by the system if the `run()` method is excepted. Using this the program logic can determine if the `run()` method completed normally or had its control asynchronously transferred to its caller.

Parameters:

`exception` - Used to invoke methods on `AsynchronouslyInterruptedException`$_{226}$ from within the `interruptAction()` method.

run

`public void` **`run`**`(AsynchronouslyInterruptedException`$_{226}$ `exception) throws AsynchronouslyInterruptedException`

The main piece of code that is executed when an implemention is given to `doInterruptible()`. When you create a class that implements this interface (usually through an anonymous inner class) you must remember to include the `throws` clause to make the method interruptible. If the throws clause is omitted the `run()` method will not be interruptible.

Parameters:

`exception` - Used to invoke methods on `AsynchronouslyInterruptedException`$_{226}$ from within the `run()` method.

Throws:

`AsynchronouslyInterruptedException`$_{226}$

11.5 AsynchronouslyInterruptedException

Declaration

```
public class AsynchronouslyInterruptedException extends
            java.lang.InterruptedException
```

All Implemented Interfaces: `java.io.Serializable`

Direct Known Subclasses: `Timed`$_{230}$

Description

An special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a `RealtimeThread`25 .

When a method is declared with `AsynchronouslyInterruptedException` in its `throws` clause the platform is expected to asynchronously throw this exception if `RealtimeThread.interrupt()` is called while the method is executing, or if such an interrupt is pending any time control returns to the method. The interrupt is *not* thrown while any methods it invokes are executing, unless they are, in turn, declared to throw the exception. This is intended to allow long-running computations to be terminated without the overhead or latency of polling with `java.lang.Thread.interrupted()` .

The `throws AsynchronouslyInterruptedException` clause is a marker on a stack frame which allows a method to be statically marked as asynchronously interruptible. Only methods that are marked this way can be interrupted.

When `Thread.interrupt()`, `RealtimeThread.interrupt()`30 , or `this.fire()` is called, the `AsynchronouslyInterruptedException` is compared against any currently pending `AsynchronouslyInterruptedException` on the thread. 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—it becomes the currently pending interrupt. Otherwise, it is discarded.

If the current method is interruptible, the exception is thrown on the thread. Otherwise, it just remains pending until control returns to an interruptible method, at which point the `AsynchronouslyInterruptedException` is thrown. When an interrupt is caught, the caller should 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 from the thread. Otherwise, it will continue to propagate outward.

`Thread.interrupt()` and `RealtimeThread.interrupt()` generate a system available generic `AsynchronouslyInterruptedException` which will always propagate outward through interruptible methods until the generic `AsynchronouslyInterruptedException` is identified and stopped. Other sources (e.g., `this.fire()` and `Timed`230) will generate a specific instance of `AsynchronouslyInterruptedException` which applications can identify and thus limit propogation.

11.5.1 Constructors

AsynchronouslyInterruptedException

`public` **`AsynchronouslyInterruptedException`**`()`

Create an instance of `AsynchronouslyInterruptedException`.

11.5.2 Methods

disable

`public boolean` **`disable`**`()`

Defer the throwing of this exception. If `RealtimeThread.interrupt()`30 is called when this exception is disabled, the exception is put in pending state. The exception will be thrown if this exception is subsequently enabled. This is valid only within a call to `doInterruptible()`. Otherwise it returns false and does nothing.

Returns: True if `this` is disabled and invoked within a call to `doInterruptible()`. False if `this` is enabled and invoked within a call to `doInterruptible()` or invoked outside of a call to `doInterruptible()`.

doInterruptible

`public boolean` **`doInterruptible`**`(Interruptible`225 `logic)`

Execute the `run()` method of the given `Interruptible`225. This method may be on the stack in exactly one `RealtimeThread`25. An attempt to invoke this method in a thread while it is on the stack of another or the same thread will cause an immediate return with a value of false.

Parameters:

`logic` - An instance of an `Interruptible`225 whose `run()` method will be called.

Returns: True if the method call completed normally. False if another call to `doInterruptible` has not completed.

enable

`public boolean` **`enable`**`()`

Enable the throwing of this exception. This method is valid only within a call to `doInterruptible()`. If invoked outside of a call to `doInterruptible()` this method returns false and does nothing.

Returns: True if `this` is enabled and invoked within a call to `doInterruptible()`. False if `this` is disable and invoked within a call to `doInterruptible()` or invoked outside of a call to `doInterruptible()`.

fire

`public boolean fire()`

Make this exception the current exception if `doInterruptible()` has been invoked and not completed.

Returns: True if `this` was fired. False if there is no current invocation of `doInterruptible()` (with no other effect), if there is already a current `doInterruptible()`, or if `disable()` has been called.

getGeneric

`public static AsynchronouslyInterruptedException`$_{226}$ `getGeneric()`

Gets the system generic `AsynchronouslyInterruptedException`, which is generated when `RealtimeThread.interrupt()`$_{30}$ is invoked.

Returns: The generic `AsynchronouslyInterruptedException`.

happened

`public boolean happened(boolean propagate)`

Used with an instance of this exception to see if the current exception is this exception.

Parameters:

`propagate` - If true and this exception is not the current one propogate the exception. If false, then the state of this is set to nonpending (i.e., it will stop propagating).

Returns: True if this is the current exception. False if this is not the current exception.

isEnabled

`public boolean isEnabled()`

Query the enabled status of this exception.

Returns: True if this is enabled. False otherwise.

propagate

`public static void **propagate**()`

Cause the pending exception to continue up the stack.

11.6 Timed

Declaration

`public class Timed extends AsynchronouslyInterruptedException`$_{226}$

All Implemented Interfaces: `java.io.Serializable`

Description

Create a scope in a `RealtimeThread`$_{25}$ for which `interrupt()` will be called 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()`. All memory use of `Timed` occurs during construction or the first invocation of `doInterruptible()`. Subsequent invokes of `doInterruptible()` do not allocate memory.

Usage: `new Timed(T).doInterruptible(interruptible);`

11.6.1 Constructors

Timed

```
public Timed(HighResolutionTime172 time)
      throws IllegalArgumentException
```

Create an instance of `Timed` with a timer set to timeout. If the time is in the past the `AsynchronouslyInterruptedException`$_{226}$ mechanism is immediately activated.

Parameters:

`time` - The interval of time between the invocation of `doInterruptible()` and when `interrupt()` is called on `currentRealtimeThread()`. If null the `java.lang.IllegalArgumentException` is thrown.

Throws:
`java.lang.IllegalArgumentException`

11.6.2 Methods

doInterruptible

`public boolean **doInterruptible**(Interruptible225 logic)`

This method will be private in 1.0.1 Execute a timeout method. Starts the timer and executes the `run()` method of the given `Interruptible`225 object.

Overrides: `doInterruptible`228 in class `AsynchronouslyInterruptedException`226

Parameters:
`logic` - Implements an `Interruptible`225 `run()` method. If null nothing happens.

resetTime

`public void **resetTime**(HighResolutionTime172 time)`

To reschedule the timeout for the next invocation of `doInterruptible()`.

Parameters:
`time` - This can be an absolute time or a relative time. If null the timeout is not changed.

CHAPTER 12 System and Options

This section contains classes that:

- Provide a common idiom for binding POSIX signals to instances of `AsyncEvent` 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. <u>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 underly-</u>

ing 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 `AsyncEvent`207 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.

12.1.1 Fields

SIGABRT

`public static final int` **`SIGABRT`**

Used by abort, replace SIGIOT in the future.

SIGALRM

`public static final int` **`SIGALRM`**

Alarm clock.

SIGBUS

```
public static final int SIGBUS
```

Bus error.

SIGCANCEL

```
public static final int SIGCANCEL
```

Thread cancellation signal used by libthread.

SIGCHLD

```
public static final int SIGCHLD
```

Child status change alias (POSIX).

SIGCLD

```
public static final int SIGCLD
```

Child status change.

SIGCONT

```
public static final int SIGCONT
```

Stopped process has been continued.

SIGEMT

```
public static final int SIGEMT
```

EMT instruction.

SIGFPE

```
public static final int SIGFPE
```

Floating point exception.

SIGFREEZE

```
public static final int SIGFREEZE
```

Special signal used by CPR.

SIGHUP

```
public static final int SIGHUP
```

Hangup.

SIGILL

`public static final int SIGILL`

Illegal instruction (not reset when caught).

SIGINT

`public static final int SIGINT`

Interrupt (rubout).

SIGIO

`public static final int SIGIO`

Socket I/O possible (SIGPOLL alias).

SIGIOT

`public static final int SIGIOT`

IOT instruction.

SIGKILL

`public static final int SIGKILL`

Kill (cannot be caught or ignored).

SIGLOST

`public static final int SIGLOST`

Resource lost (e.g., record-lock lost).

SIGLWP

`public static final int SIGLWP`

Special signal used by thread library.

SIGPIPE

`public static final int SIGPIPE`

Write on a pipe with no one to read it.

SIGPOLL

`public static final int SIGPOLL`

Pollable event occured.

SIGPROF

`public static final int SIGPROF`

Profiling timer expired.

SIGPWR

`public static final int SIGPWR`

Power-fail restart.

SIGQUIT

`public static final int SIGQUIT`

Quit (ASCII FS).

SIGSEGV

`public static final int SIGSEGV`

Segmentation violation.

SIGSTOP

`public static final int SIGSTOP`

Stop (cannot be caught or ignored).

SIGSYS

`public static final int SIGSYS`

Bad argument to system call.

SIGTERM

`public static final int SIGTERM`

Software termination signal from kill.

SIGTHAW

`public static final int SIGTHAW`

Special signal used by CPR.

SIGTRAP

`public static final int SIGTRAP`

Trace trap (not reset when caught).

SIGTSTP

`public static final int SIGTSTP`

User stop requested from tty.

SIGTTIN

`public static final int SIGTTIN`

Background tty read attempted.

SIGTTOU

`public static final int SIGTTOU`

Background tty write attempted.

SIGURG

`public static final int SIGURG`

Urgent socket condition.

SIGUSR1

`public static final int SIGUSR1`

User defined signal = 1.

SIGUSR2

`public static final int SIGUSR2`

User defined signal = 2.

SIGVTALRM

`public static final int SIGVTALRM`

Virtual timer expired.

SIGWAITING

`public static final int` **`SIGWAITING`**

Process's lwps are blocked.

SIGWINCH

`public static final int` **`SIGWINCH`**

Window size change.

SIGXCPU

`public static final int` **`SIGXCPU`**

Exceeded cpu limit.

SIGXFSZ

`public static final int` **`SIGXFSZ`**

Exceeded file size limit.

12.1.2 Constructors

POSIXSignalHandler

`public` **`POSIXSignalHandler`**`()`

12.1.3 Methods

addHandler

`public static void` **`addHandler`**`(int signal, AsyncEventHandler`$_{210}$ `handler)`

Add the given `AsyncEventHandler`$_{210}$ to the list of handlers of the `AsyncEvent`$_{207}$ 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 `AsyncEventHandler`$_{210}$ which will be scheduled when the given signal occurs.

removeHandler

```
public static void removeHandler(int signal,
     AsyncEventHandler210 handler)
```

Remove the given `AsyncEventHandler`$_{210}$ to the list of handlers of the `AsyncEvent`$_{207}$ 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 `AsyncEventHandler`$_{210}$ which will be scheduled when the given signal occurs.

setHandler

```
public static void setHandler(int signal,
     AsyncEventHandler210 handler)
```

Set the given `AsyncEventHandler`$_{210}$ as the handler of the `AsyncEvent`$_{207}$ 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 `AsyncEventHandler`$_{210}$ 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).

12.2 RealtimeSecurity

Declaration

`public class **RealtimeSecurity**`

Description

Security policy object for real-time specific issues. Primarily used to control access to physical memory.

12.2.1 Constructors

RealtimeSecurity

```
public RealtimeSecurity()
```

Create an `RealtimeSecurity` object.

12.2.2 Methods

checkAccessPhysical

```
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.

checkAccessPhysicalRange

```
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.

checkSetFilter

```
public void checkSetFilter()
     throws SecurityException
```

Checks whether the application is allowed to set filter objects.

Throws:

`java.lang.SecurityException` - The application doesn't have permission to set filter objects.

checkSetScheduler

```
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

BIG_ENDIAN

`public static final byte BIG_ENDIAN`

Value to set the byte ordering for the underlying hardware.

BYTE_ORDER

`public static final byte BYTE_ORDER`

The byte ordering of the underlying hardware.

LITTLE_ENDIAN

`public static final byte LITTLE_ENDIAN`

Value to set the byte ordering for the underlying hardware.

12.3.2 Constructors

RealtimeSystem

`public RealtimeSystem()`

12.3.3 Methods

currentGC

`public static GarbageCollector`$_{154}$ `currentGC()`

Return a reference to the currently active garbage collector for the heap.

Returns: A `GarbageCollector`$_{154}$ object which is the current collector collecting objects on the traditional Java heap.

getConcurrentLocksUsed

`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.

getMaximumConcurrentLocks

`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.

Returns: An integer whose value is the maximum number of locks that can be in simultaneous use.

getSecurityManager

`public static RealtimeSecurity`$_{240}$ `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`$_{240}$ object representing the default real-time security manager.

setMaximumConcurrentLocks

```
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:

`number` - 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.

setMaximumConcurrentLocks

```
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.

`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`$_{246}$ is thrown.

setSecurityManager

```
public static void setSecurityManager(RealtimeSecurity240
      manager)
```

Sets a new real-time security manager.

Parameters:

`manager` - A `RealtimeSecurity`$_{240}$ 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 classes that:

- Add additional exception classes required by the entirety of the other sections of this specification.
- Provide for the ability to asynchronously transfer the control of program logic.

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.
3. Instances of the class `AsynchronouslyInterruptedException` can be generated by execution of program logic and by internal virtual machine mechanisms that are asynchronous to the execution of program logic which is the target of the exception.
4. Program logic that exists in methods that throw `AsynchronouslyInterrupted-Exception` is subject to receiving an instance of `AsynchronouslyInterrupted-Exception` at any time during execution.

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 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.

13.1.1 Constructors

ResourceLimitError

`public` **`ResourceLimitError`**`()`

A constructor for `ResourceLimitError`.

ResourceLimitError

`public` **`ResourceLimitError`**`(java.lang.String description)`

A descriptive constructor for `ResourceLimitError`.

Parameters:

`description` - The description of the exception.

13.2 OffsetOutOfBoundsException

Declaration

`public class` **`OffsetOutOfBoundsException`** `extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown if the constructor of an `ImmortalPhysicalMemory`113 , `LTPhysicalMemory`122 , `VTPhysicalMemory`129 , `RawMemoryAccess`135 , or `RawMemoryFloatAccess`145 is given an invalid address.

13.2.1 Constructors

OffsetOutOfBoundsException

`public OffsetOutOfBoundsException()`

A constructor for `OffsetOutOfBoundsException`.

OffsetOutOfBoundsException

`public OffsetOutOfBoundsException(java.lang.String description)`

A descriptive constructor for `OffsetOutOfBoundsException`.

Parameters:

`description` - A description of the exception.

13.3 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 `MemoryArea`90 . For example this will be thrown if logic in a `NoHeapRealtimeThread`38 attempts to refer to an object in the traditional Java heap.

13.3.1 Constructors

MemoryAccessError

`public MemoryAccessError()`

A constructor for `MemoryAccessError`.

MemoryAccessError

```
public MemoryAccessError(java.lang.String description)
```

A descriptive constructor for `MemoryAccessError`.

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 if logic attempts to assign a reference to an object in scoped memory (an area of memory identified be an instance of `ScopedMemory`$_{98}$ to a field of an object in immortal memory.

13.4.1 Constructors

IllegalAssignmentError

```
public IllegalAssignmentError()
```

A constructor for `IllegalAssignmentError`.

IllegalAssignmentError

```
public IllegalAssignmentError(java.lang.String description)
```

A descriptive constructor for `IllegalAssignmentError`.

Parameters:

`description` - Description of the error.

13.5 SizeOutOfBoundsException

Declaration

```
public class SizeOutOfBoundsException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown if the constructor of an `ImmortalPhysicalMemory`$_{113}$, `LTPhysicalMemory`$_{122}$, `VTPhysicalMemory`$_{129}$, `RawMemoryAccess`$_{135}$, or `RawMemoryFloatAccess`$_{145}$ is given an invalid size or if an accessor method on one of the above classes would cause access to an invalid address.

13.5.1 Constructors

SizeOutOfBoundsException

`public` **`SizeOutOfBoundsException`**`()`

A constructor for `SizeOutOfBoundsException`.

SizeOutOfBoundsException

`public` **`SizeOutOfBoundsException`**`(java.lang.String description)`

A descriptive constructor for `SizeOutOfBoundsException`.

Parameters:

`description` - The description of the exception.

13.6 UnsupportedPhysicalMemoryException

Declaration

```
public class UnsupportedPhysicalMemoryException extends
            java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown when the underlying hardware does not support the type of physical memory given to the physical memory `create()` method.

See Also: `RawMemoryAccess`$_{135}$, `RawMemoryFloatAccess`$_{145}$, `ImmortalPhysicalMemory`$_{113}$, `LTPhysicalMemory`$_{122}$, `VTPhysicalMemory`$_{129}$

13.6.1 Constructors

UnsupportedPhysicalMemoryException

`public` **`UnsupportedPhysicalMemoryException`**`()`

A constructor for `UnsupportedPhysicalMemoryException`.

UnsupportedPhysicalMemoryException

```
public UnsupportedPhysicalMemoryException(java.lang.String
     description)
```

A descriptive constructor for `UnsupportedPhysicalMemoryException`.

Parameters:

`description` - The description of the exception.

13.7 MemoryScopeException

Declaration

```
public class MemoryScopeException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

Thrown if construction of any of the wait-free queues is attempted with the ends of the queues in incompatible memory areas.

13.7.1 Constructors

MemoryScopeException

```
public MemoryScopeException()
```

A constructor for `MemoryScopeException`.

MemoryScopeException

```
public MemoryScopeException(java.lang.String description)
```

A descriptive constructor for `MemoryScopeException`.

Parameters:

`description` - A description of the exception.

13.8 ThrowBoundaryError

Declaration

```
public class ThrowBoundaryError extends java.lang.Error
```

All Implemented Interfaces: `java.io.Serializable`

Description

The error thrown by `MemoryArea.enter(Runnable)`92 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 `MemoryArea.enter(Runnable)`92 .

13.8.1 Constructors

ThrowBoundaryError

`public ThrowBoundaryError()`

A constructor for `ThrowBoundaryError`.

ThrowBoundaryError

`public ThrowBoundaryError(java.lang.String description)`

A descriptive constructor for `ThrowBoundaryError`.

Parameters:

`description` - Description of the error.

13.9 DuplicateFilterException

Declaration

`public class DuplicateFilterException extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Description

`PhysicalMemoryManager`110 can only accomodate 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.

13.9.1 Constructors

DuplicateFilterException

`public DuplicateFilterException()`

A constructor for `DuplicatefilterException`.

DuplicateFilterException

`public DuplicateFilterException(java.lang.String description)`

A descriptive constructor for `DuplicatefilterException`.

Parameters:

`description` - Description of the error.

13.10 ArrivalTimeQueueOverflowException

Declaration

```
public class ArrivalTimeQueueOverflowException extends
              java.lang.Exception
```

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 then the fire() method shall throw a this. If the arrival time is a result of a happening to which the instance of AsyncEventHandler is bound then the arrival time is ignored.

13.10.1 Constructors

ArrivalTimeQueueOverflowException

`public ArrivalTimeQueueOverflowException()`

A constructor for `ArrivalTimeQueueOverflowException`.

ArrivalTimeQueueOverflowException

```
public ArrivalTimeQueueOverflowException(java.lang.String
      description)
```

A descriptive constructor for `ArrivalTimeQueueOverflowException`.

Parameters:

`description` - A description of the exception.

13.11 MITViolationException

Declaration

```
public class MITViolationException extends java.lang.Exception
```

All Implemented Interfaces: `java.io.Serializable`

Description

This exception is thrown minimum interarrival time gets violated.

13.11.1 Constructors

MITViolationException

public **MITViolationException**()

A constructor for `MITViolationException`.

MITViolationException

public **MITViolationException**(java.lang.String description)

A descriptive constructor for `MITViolationException`.

Parameters:

`description` - Description of the error.

13.12 ScopedCycleException

Declaration

`public class ScopedCycleException extends java.lang.RuntimeException`

All Implemented Interfaces: `java.io.Serializable`

13.12.1 Constructors

ScopedCycleException

public **ScopedCycleException**()

A constructor for `ScopedCycleException`.

ScopedCycleException

public **ScopedCycleException**(java.lang.String description)

A descriptive constructor for `ScopedCycleException`.

Parameters:

`description` - Description of the error.

13.13 MemoryInUseException

Declaration

`public class` **`MemoryInUseException`** `extends java.lang.RuntimeException`

All Implemented Interfaces: `java.io.Serializable`

Description

There has been attempt to allocation a range of physical or virtual memory that is already in use.

13.13.1 Constructors

MemoryInUseException

`public` **`MemoryInUseException`**`()`

A constructor for `MemoryInUseException`.

MemoryInUseException

`public` **`MemoryInUseException`**`(java.lang.String description)`

A descriptive constructor for `MemoryInUseException`.

Parameters:

`description` - Description of the error.

13.14 MemoryTypeConflictException

Declaration

`public class` **`MemoryTypeConflictException`** `extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Description

This exception is thrown when the `PhysicalMemoryManager`$_{110}$ 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.

13.14.1 Constructors

MemoryTypeConflictException

`public` **`MemoryTypeConflictException`**`()`

A constructor for `MemoryTypeConflictException`.

MemoryTypeConflictException

`public` **`MemoryTypeConflictException`**`(java.lang.String description)`

A descriptive constructor for `MemoryTypeConflictException`.

Parameters:

`description` - A description of the exception.

13.15 InaccessibleAreaException

Declaration

`public class` **`InaccessibleAreaException`** `extends java.lang.Exception`

All Implemented Interfaces: `java.io.Serializable`

Description

The specified memory area is not above the current allocation context on the current thread scope stack.

13.15.1 Constructors

InaccessibleAreaException

`public` **`InaccessibleAreaException`**`()`

A constructor for `InaccessibleAreaException`.

InaccessibleAreaException

`public` **`InaccessibleAreaException`**`(java.lang.String description)`

A descriptive constructor for `InaccessibleAreaException`.

Parameters:

`description` - Description of the error.

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 `AsyncEvent`$_{207}$ attempts to bind to a happening that does not exist.

13.16.1 Constructors

UnknownHappeningException

`public UnknownHappeningException()`

A constructor for `UnknownHappeningException`.

UnknownHappeningException

`public UnknownHappeningException(java.lang.String description)`

A descriptive constructor for `UnknownHappeningException`.

Parameters:

`description` - Description of the error.

CHAPTER 14

Quick Reference Guide

Interface Summary	
Interruptible	`Interruptible` is an interface implemented by classes that will be used as arguments on the `doInterruptible()` of `Asynchronously-InterruptedException` and its subclasses.
PhysicalMemoryTypeFilter	
Schedulable	Handlers and other objects can be run by a `Scheduler` if they provide a `run()` method and the methods defined below.

Class Summary	
AbsoluteTime	An object that represents a specific point in time given by milliseconds plus nanoseconds past the epoch (January 1, 1970, 00:00:00 GMT).
AperiodicParameters	This release parameter object characterizes a schedulable object that may become active at any time.
AsyncEvent	An asynchronous event represents something that can happen, like a light turning red.
AsyncEventHandler	An asynchronous event handler encapsulates code that gets run at some time after an `AsyncEvent` occurs.

BoundAsyncEventHandler	A bound asynchronous event handler is an asynchronous event handler that is permanently bound to a thread.
Clock	A clock advances from the past, through the present, into the future.
GarbageCollector	The system shall provide dynamic and static information characterizing the temporal behavior and imposed overhead of any garbage collection algorithm provided by the system.
HeapMemory	The `HeapMemory` class is a singleton object that allows logic within other scoped memory to allocate objects in the Java heap.
HighResolutionTime	Class `HighResolutionTime` is the base class for AbsoluteTime, RelativeTime, RationalTime.
ImmortalMemory	`ImmortalMemory` is a memory resource that is shared among all threads.
ImmortalPhysicalMemory	An instance of `ImmortalPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type.
ImportanceParameters	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.
LTMemory	`LTMemory` represents a memory area, allocated per `RealtimeThread`, or for a group of real-time threads, guaranteed by the system to have linear time allocation.
LTPhysicalMemory	An instance of `LTPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type.
MemoryArea	`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.
MemoryParameters	Memory parameters can be given on the constructor of `RealtimeThread` and `AsyncEventHandler`.
MonitorControl	Abstract superclass for all monitor control policy objects.

NoHeapRealtimeThread	A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`.
OneShotTimer	A timed AsyncEvent that is driven by a clock.
PeriodicParameters	This release parameter indicates that the `RealtimeThread.waitForNextPeriod()` method on the associated `Schedulable` object will be unblocked at the start of each period.
PeriodicTimer	An AsyncEvent whose fire method is executed periodically according to the given parameters.
PhysicalMemoryManager	The `PhysicalMemoryManager` is available for use by the various physical memory accessor objects (`LTPhysicalMemory`, `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.
POSIXSignalHandler	Use instances of `AsyncEvent` to handle POSIX signals.
PriorityCeilingEmulation	Monitor control class specifying use of the priority ceiling emulation protocol for monitor objects.
PriorityInheritance	Monitor control class specifying use of the priority inheritance protocol for object monitors.
PriorityParameters	Instances of this class should be assigned to threads that are managed by schedulers which use a single integer to determine execution order.
PriorityScheduler	
ProcessingGroupParameters	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.
RationalTime	An object that represents a time interval millis/1E3+nanos/1E9 seconds long that is divided into subintervals by some frequency.
RawMemoryAccess	An instance of `RawMemoryAccess` models a range of physical memory as a fixed sequence of bytes.
RawMemoryFloatAccess	This class holds the accessor methods for accessing a raw memory area by float and double types.
RealtimeSecurity	Security policy object for real-time specific issues.

RealtimeSystem	`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.
RealtimeThread	`RealtimeThread` extends `Thread` and includes classes and methods to get and set parameter objects, manage the execution of those threads with a `PeriodicParameters`, and waiting.
RelativeTime	An object that represents a time interval millis/ 1E3+nanos/1E9 seconds long.
ReleaseParameters	The abstract top-level class for release characteristics of threads.
Scheduler	An instance of `Scheduler` manages the execution of schedulable objects and may implement a feasibility algorithm.
SchedulingParameters	Subclasses of `SchedulingParameters` (`PriorityParameters`, `ImportanceParameters`, and any others defined for particular schedulers) provide the parameters to be used by the `Scheduler`.
ScopedMemory	`ScopedMemory` is the abstract base class of all classes dealing with representations of memory spaces with a limited lifetime.
SizeEstimator	This is a convenient class to help people figure out how much memory they need.
SporadicParameters	A notice to the scheduler that the associated schedulable object's run method will be released aperiodically but with a minimum time between releases.
Timer	A Timer is a timed event that measures time relative to a given Clock.
VTMemory	The execution time of an allocation from a `VTMemory` area may take a variable amount of time.
VTPhysicalMemory	An instance of `VTPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type.
WaitFreeDequeue	The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` and `Thread`.

WaitFreeReadQueue	The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` and `Thread`.
WaitFreeWriteQueue	The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` and `Thread`.

Exception Summary	
AsynchronouslyInterruptedException	An special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a `RealtimeThread`.
DuplicateFilterException	`PhysicalMemoryManager` can only accommodate one filter object for each type of memory.
InaccessibleAreaException	The specified memory area is not above the current allocation context on the current thread scope stack.
MemoryInUseException	Thrown when an attempt is made to allocate a range of physical or virtual memory that is already in use.
MemoryScopeException	Thrown if construction of any of the wait-free queues is attempted with the ends of the queues in incompatible memory areas.
MemoryTypeConflictException	This exception is thrown when the `PhysicalMemoryManager` is given conflicting specifications for memory.
MITViolationException	Thrown by the `fire()` method of an instance of `AsyncEvent` when the bound instance of `ReleaseParameters` type of `SporadicParameters` has `mitViolationExcept` behavior and the minimum interarrival time gets violated.
OffsetOutOfBoundsException	Thrown if the constructor of an `ImmortalPhysicalMemory`, `RawMemoryAccess`, or `RawMemoryFloatAccess` is given an invalid address.
ScopedCycleException	Thrown when a user tries to enter a `ScopedMemory` that is already accessible (`ScopedMemory` is present on stack) or when a user tries to create `ScopedMemory` cycle spanning threads (tries to make cycle in the VM `ScopedMemory` tree structure).

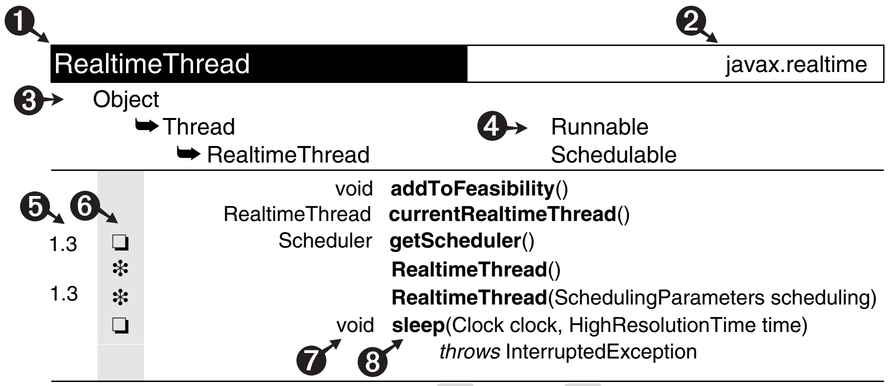
SizeOutOfBoundsException	Thrown if the constructor of an `ImmortalPhysicalMemory`, `RawMemoryAccess`, or `RawMemoryFloatAccess` is given an invalid size or if an accessor method on one of the above classes would cause access to an invalid address.
Timed	Create a scope in a `RealtimeThread` for which `interrupt()` will be called at the expiration of a timer.
UnknownHappeningException	Thrown when `bindTo()` is called with an illegal `happening.`
UnsupportedPhysicalMemoryException	Thrown when the underlying hardware does not support the type of physical memory given to the physical memory `create()` method.

Error Summary	
IllegalAssignmentError	The exception thrown on an attempt to make an illegal assignment.
MemoryAccessError	This error is thrown on an attempt to refer to an object in an inaccessible `MemoryArea`.
ResourceLimitError	Thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks.
ThrowBoundaryError	The error thrown by `public void enter(Runnable logic)` 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 `public void enter(Runnable logic)`.

ALMANAC LEGEND

The almanac presents classes and intefaces 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*.



		void	**addToFeasibility**()
		RealtimeThread	**currentRealtimeThread**()
1.3	❑	Scheduler	**getScheduler**()
	❇		**RealtimeThread**()
1.3	❇		**RealtimeThread**(SchedulingParameters scheduling)
	❑	void	**sleep**(Clock clock, HighResolutionTime time) *throws* InterruptedException

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 does not appear, the member is public. (Private and package-private modifiers also have no symbols.) One symbol from each group can appear in this column.

Modifiers		**Access Modifiers**	**Constructors and Fields**	
❍	abstract	♦protected	❇	constructor
●	final		✍	field
❑	static			
■	static final			

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.

CHAPTER 15

Almanac

AbsoluteTime — javax.realtime

```
Object
  ➥HighResolutionTime                Comparable
    ➥AbsoluteTime
```

	AbsoluteTime	**absolute(Clock clock)**
	AbsoluteTime	**absolute(Clock clock, AbsoluteTime destination)**
❇		**AbsoluteTime()**
❇		**AbsoluteTime(AbsoluteTime time)**
❇		**AbsoluteTime(java.util.Date date)**
❇		**AbsoluteTime(long millis, int nanos)**
	AbsoluteTime	**add(long millis, int nanos)**
	AbsoluteTime	**add(long millis, int nanos, AbsoluteTime destination)**
●	AbsoluteTime	**add(RelativeTime time)**
	AbsoluteTime	**add(RelativeTime time, AbsoluteTime destination)**
	java.util.Date	**getDate()**
	RelativeTime	**relative(Clock clock)**
	RelativeTime	**relative(Clock clock, RelativeTime destination)**
	void	**set(java.util.Date date)**
●	RelativeTime	**subtract(AbsoluteTime time)**
●	RelativeTime	**subtract(AbsoluteTime time, RelativeTime destination)**
●	AbsoluteTime	**subtract(RelativeTime time)**
	AbsoluteTime	**subtract(RelativeTime time, AbsoluteTime destination)**
	String	**toString()**

AperiodicParameters — javax.realtime

```
Object
  ➥ReleaseParameters
    ➥AperiodicParameters
```

❇		**AperiodicParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)**
	boolean	**setIfFeasible(RelativeTime cost, RelativeTime deadline)**

ArrivalTimeQueueOverflowException — javax.realtime

```
Object
  ➥Throwable                                  java.io.Serializable
    ➥Exception
      ➥ArrivalTimeQueueOverflowException
```

❇		**ArrivalTimeQueueOverflowException()**
❇		**ArrivalTimeQueueOverflowException(String description)**

AsyncEvent — javax.realtime

```
Object
  ➥AsyncEvent
```

	void	**addHandler(AsyncEventHandler handler)**
❇		**AsyncEvent()**
	void	**bindTo(String happening)** ***throws*** **UnknownHappeningException**
	ReleaseParameters	**createReleaseParameters()**
	void	**fire()**
	boolean	**handledBy(AsyncEventHandler handler)**
	void	**removeHandler(AsyncEventHandler handler)**
	void	**setHandler(AsyncEventHandler handler)**
	void	**unbindTo(String happening)** ***throws*** **UnknownHappeningException**

AsyncEventHandler javax.realtime

Object
➥AsyncEventHandler Schedulable

	boolean	**addIfFeasible()**
	boolean	**addToFeasibility()**
❇		**AsyncEventHandler()**
❇		**AsyncEventHandler(boolean nonheap)**
❇		**AsyncEventHandler(Runnable logic)**
❇		**AsyncEventHandler(Runnable logic, boolean nonheap)**
❇		**AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap)**
❇		**AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic)**
❇		**AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic, boolean nonheap)**
●♦	int	**getAndClearPendingFireCount()**
♦	int	**getAndDecrementPendingFireCount()**
♦	int	**getAndIncrementPendingFireCount()**
	MemoryArea	**getMemoryArea()**
	MemoryParameters	**getMemoryParameters()**
●♦	int	**getPendingFireCount()**
	ProcessingGroupParameters	**getProcessingGroupParameters()**
	ReleaseParameters	**getReleaseParameters()**
	Scheduler	**getScheduler()**
	SchedulingParameters	**getSchedulingParameters()**
	void	**handleAsyncEvent()**
	boolean	**removeFromFeasibility()**
●	void	**run()**
	boolean	**setIfFeasible(ReleaseParameters release, MemoryParameters memory)**

	boolean	**setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)**
	boolean	**setIfFeasible(ReleaseParameters release, 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)** ***throws*** **IllegalThreadStateException**
	void	**setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)** ***throws*** **IllegalThreadStateException**
	void	**setSchedulingParameters(SchedulingParameters scheduling)**
	boolean	**setSchedulingParametersIfFeasible(SchedulingParameters scheduling)**

AsynchronouslyInterruptedException — javax.realtime

```
Object
  ➥Throwable                              java.io.Serializable
      ➥Exception
          ➥InterruptedException
              ➥AsynchronouslyInterruptedException
```

❇		**AsynchronouslyInterruptedException()**
	boolean	**disable()**
	boolean	**doInterruptible(Interruptible 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, Runnable logic, boolean nonheap)**

Clock javax.realtime

```
Object
    ➥Clock
```

❈		**Clock()**
❏	Clock	**getRealtimeClock()**
❍	RelativeTime	**getResolution()**
	AbsoluteTime	**getTime()**
❍	void	**getTime(AbsoluteTime time)**
❍	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
```

❏	HeapMemory	**instance()**
	long	**memoryConsumed()**
	long	**memoryRemaining()**

HighResolutionTime javax.realtime

```
Object
  ➥HighResolutionTime                    Comparable
```

❍	AbsoluteTime	**absolute(Clock clock)**
❍	AbsoluteTime	**absolute(Clock clock, AbsoluteTime dest)**
	int	**compareTo(HighResolutionTime time)**
	int	**compareTo(Object object)**
	boolean	**equals(HighResolutionTime time)**
	boolean	**equals(Object object)**
●	long	**getMilliseconds()**
●	int	**getNanoseconds()**
	int	**hashCode()**
❍	RelativeTime	**relative(Clock clock)**
❍	RelativeTime	**relative(Clock clock, HighResolutionTime dest)**
	void	**set(HighResolutionTime time)**
	void	**set(long millis)**
	void	**set(long millis, int nanos)**
❏	void	**waitForObject(Object target, HighResolutionTime time)** ***throws*** **InterruptedException**

IllegalAssignmentError javax.realtime

```
Object
  ➥Throwable                    java.io.Serializable
    ➥Error
      ➥IllegalAssignmentError
```

❇		**IllegalAssignmentError()**
❇		**IllegalAssignmentError(String description)**

ImmortalMemory javax.realtime

Object
  ➥MemoryArea
    ➥ImmortalMemory

❏	ImmortalMemory	**instance()**

ImmortalPhysicalMemory javax.realtime

Object
  ➥MemoryArea
    ➥ImmortalPhysicalMemory

❇	**ImmortalPhysicalMemory(Object type, long size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇	**ImmortalPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇	**ImmortalPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇	**ImmortalPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇	**ImmortalPhysicalMemory(Object type, long base, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**

❇		**ImmortalPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**ImmortalPhysicalMemory(Object type, SizeEstimator size)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**ImmortalPhysicalMemory(Object type, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**

ImportanceParameters javax.realtime

Object
➥SchedulingParameters
➥PriorityParameters
➥ImportanceParameters

	int	**getImportance()**
❇		**ImportanceParameters(int priority, int importance)**
	void	**setImportance(int importance)**
	String	**toString()**

InaccessibleAreaException javax.realtime

Object
➥Throwable java.io.Serializable
➥Exception
➥InaccessibleAreaException

❇		**InaccessibleAreaException()**
❇		**InaccessibleAreaException(String description)**

Interruptible javax.realtime

Interruptible

	void	**interruptAction(AsynchronouslyInterruptedException exception)**
	void	**run(AsynchronouslyInterruptedException exception)** ***throws*** **AsynchronouslyInterruptedException**

LTMemory javax.realtime

```
Object
    ➥MemoryArea
        ➥ScopedMemory
            ➥LTMemory
```

	long	**getMaximumSize()**
❇		**LTMemory(long initialSizeInBytes, long maxSizeInBytes)**
❇		**LTMemory(long initialSizeInBytes, long maxSizeInBytes, 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) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**LTPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**LTPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**LTPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**

❇		**LTPhysicalMemory(Object type, long base, SizeEstimator size)** ***throws*** **SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**LTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**LTPhysicalMemory(Object type, SizeEstimator size)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**LTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
	String	**toString()**

MemoryAccessError — javax.realtime

Object
  ➥Throwable — java.io.Serializable
    ➥Error
      ➥MemoryAccessError

❇		**MemoryAccessError()**
❇		**MemoryAccessError(String description)**

MemoryArea — javax.realtime

Object
  ➥MemoryArea

	void	**enter()** ***throws*** **ScopedCycleException**
	void	**enter(Runnable logic)** ***throws*** **ScopedCycleException**
	void	**executeInArea(Runnable logic)** ***throws*** **InaccessibleAreaException**
❏	MemoryArea	**getMemoryArea(Object object)**
❇♦		**MemoryArea(long sizeInBytes)**
❇♦		**MemoryArea(long sizeInBytes, Runnable logic)**
❇♦		**MemoryArea(SizeEstimator size)**
❇♦		**MemoryArea(SizeEstimator size, Runnable logic)**

	long	**memoryConsumed()**
	long	**memoryRemaining()**
	Object	**newArray(Class type, int number)** ***throws*** **IllegalAccessException, InstantiationException**
	Object	**newInstance(Class type)** ***throws*** **IllegalAccessException, InstantiationException**
	Object	**newInstance(reflect.Constructor c, Object args)** ***throws*** **IllegalAccessException, InstantiationException**
	long	**size()**

MemoryInUseException javax.realtime

Object
➥Throwable java.io.Serializable
➥Exception
➥RuntimeException
➥MemoryInUseException

❇		**MemoryInUseException()**
❇		**MemoryInUseException(String description)**

MemoryParameters javax.realtime

Object
➥MemoryParameters

	long	**getAllocationRate()**
	long	**getMaxImmortal()**
	long	**getMaxMemoryArea()**
❇		**MemoryParameters(long maxMemoryArea, long maxImmortal)** ***throws*** **IllegalArgumentException**
❇		**MemoryParameters(long maxMemoryArea, long maxImmortal, long allocationRate)** ***throws*** **IllegalArgumentException**
✍■	long	**NO_MAX**
	void	**setAllocationRate(long allocationRate)**
	boolean	**setAllocationRateIfFeasible(int allocationRate)**
	boolean	**setMaxImmortalIfFeasible(long maximum)**
	boolean	**setMaxMemoryAreaIfFeasible(long maximum)**

MemoryScopeException — javax.realtime

```
Object
   ➥Throwable                          java.io.Serializable
      ➥Exception
         ➥MemoryScopeException
```

❇		**MemoryScopeException()**
❇		**MemoryScopeException(String description)**

MemoryTypeConflictException — javax.realtime

```
Object
   ➥Throwable                          java.io.Serializable
      ➥Exception
         ➥MemoryTypeConflictException
```

❇		**MemoryTypeConflictException()**
❇		**MemoryTypeConflictException(String description)**

MITViolationException — javax.realtime

```
Object
   ➥Throwable                          java.io.Serializable
      ➥Exception
         ➥MITViolationException
```

❇		**MITViolationException()**
❇		**MITViolationException(String description)**

MonitorControl — javax.realtime

```
Object
   ➥MonitorControl
```

❏	MonitorControl	**getMonitorControl()**
❏	MonitorControl	**getMonitorControl(Object monitor)**
❇		**MonitorControl()**
❏	void	**setMonitorControl(MonitorControl policy)**
❏	void	**setMonitorControl(Object monitor, MonitorControl policy)**

NoHeapRealtimeThread javax.realtime

```
Object
    ➥Thread                                   Runnable
        ➥RealtimeThread                       Schedulable
            ➥NoHeapRealtimeThread
```

```
❇        NoHeapRealtimeThread(SchedulingParameters sched
             uling, MemoryArea area)
             throws IllegalArgumentException
❇        NoHeapRealtimeThread(SchedulingParameters sched
             uling, ReleaseParameters release,
             MemoryArea area)
             throws IllegalArgumentException
❇        NoHeapRealtimeThread(SchedulingParameters sched
             uling, ReleaseParameters release,
             MemoryParameters memory, MemoryArea area,
             ProcessingGroupParameters group,
             Runnable logic)
             throws IllegalArgumentException
    void start()
```

OffsetOutOfBoundsException javax.realtime

```
Object
    ➥Throwable                                java.io.Serializable
        ➥Exception
            ➥OffsetOutOfBoundsException
```

```
❇        OffsetOutOfBoundsException()
❇        OffsetOutOfBoundsException(String description)
```

OneShotTimer javax.realtime

```
Object
    ➥AsyncEvent
        ➥Timer
            ➥OneShotTimer
```

```
❇        OneShotTimer(HighResolutionTime time,
             AsyncEventHandler handler)
❇        OneShotTimer(HighResolutionTime start, Clock clock,
             AsyncEventHandler handler)
```

PeriodicParameters javax.realtime

Object
➥ReleaseParameters
➥PeriodicParameters

	RelativeTime	**getPeriod()**
	HighResolutionTime	**getStart()**
❇		**PeriodicParameters(HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)**
	boolean	**setIfFeasible(RelativeTime period, RelativeTime cost, RelativeTime deadline)**
	void	**setPeriod(RelativeTime p)**
	void	**setStart(HighResolutionTime start)**

PeriodicTimer javax.realtime

Object
➥AsyncEvent
➥Timer
➥PeriodicTimer

	ReleaseParameters	**createReleaseParameters()**
	void	**fire()**
	AbsoluteTime	**getFireTime()**
	void	**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

✍■	String	**ALIGNED**
✍■	String	**BYTESWAP**
✍■	String	**DMA**
❏	boolean	**isRemovable(long address, long size)**

❏	boolean	**isRemoved(long address, long size)**
❏	void	**onInsertion(long base, long size, AsyncEventHandler aeh)**
❏	void	**onRemoval(long base, long size, AsyncEventHandler aeh)**
✻		**PhysicalMemoryManager()**
■	void	**registerFilter(Object name, PhysicalMemoryTypeFilter filter)** ***throws*** **DuplicateFilterException, IllegalArgumentException**
■	void	**removeFilter(Object name)**
✍■	String	**SHARED**

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()**
	void	**onInsertion(long base, long size, AsyncEventHandler aeh)**
	void	**onRemoval(long base, long size, AsyncEventHandler aeh)**
	long	**vFind(long base, long size)**

POSIXSignalHandler **javax.realtime**

Object
➥POSIXSignalHandler

❏	void	**addHandler(int signal, AsyncEventHandler handler)**
✻		**POSIXSignalHandler()**
❏	void	**removeHandler(int signal, AsyncEventHandler handler)**
❏	void	**setHandler(int signal, AsyncEventHandler handler)**
✍■	int	**SIGABRT**
✍■	int	**SIGALRM**
✍■	int	**SIGBUS**
✍■	int	**SIGCANCEL**
✍■	int	**SIGCHLD**

✍■	int **SIGCLD**
✍■	int **SIGCONT**
✍■	int **SIGEMT**
✍■	int **SIGFPE**
✍■	int **SIGFREEZE**
✍■	int **SIGHUP**
✍■	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

	int	**getDefaultCeiling()**
✻		**PriorityCeilingEmulation(int ceiling)**

PriorityInheritance javax.realtime

Object
➥MonitorControl
➥PriorityInheritance

❏	PriorityInheritance	**instance()**
✻		**PriorityInheritance()**

PriorityParameters javax.realtime

Object
➥SchedulingParameters
➥PriorityParameters

	int	**getPriority()**
✻		**PriorityParameters(int priority)**
	void	**setPriority(int priority)** ***throws*** **IllegalArgumentException**
	String	**toString()**

PriorityScheduler javax.realtime

Object
➥Scheduler
➥PriorityScheduler

♦	void	**addToFeasibility(Schedulable schedulable)**
	void	**fireSchedulable(Schedulable schedulable)**
	int	**getMaxPriority()**
❏	int	**getMaxPriority(Thread thread)**
	int	**getMinPriority()**
❏	int	**getMinPriority(Thread thread)**
	int	**getNormPriority()**
❏	int	**getNormPriority(Thread thread)**
	String	**getPolicyName()**

❏	PriorityScheduler	**instance()**
	boolean	**isFeasible()**
✍■	int	**MAX_PRIORITY**
✍■	int	**MIN_PRIORITY**
❇♦		**PriorityScheduler()**
♦	boolean	**removeFromFeasibility(Schedulable schedulable)**
	boolean	**setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)**
	boolean	**setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)**

ProcessingGroupParameters javax.realtime

Object
➥ProcessingGroupParameters

	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
      ➥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)** ***throws*** **IllegalArgumentException**
❇		**RationalTime(int frequency, RelativeTime interval)** ***throws*** **IllegalArgumentException**
	void	**set(long millis, int nanos)** ***throws*** **IllegalArgumentException**
	void	**setFrequency(int frequency)** ***throws*** **ArithmeticException**

RawMemoryAccess — javax.realtime

```
Object
   ➥RawMemoryAccess
```

byte	**getByte(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
void	**getBytes(long offset, byte[] bytes, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
int	**getInt(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
void	**getInts(long offset, int[] ints, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
long	**getLong(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
void	**getLongs(long offset, long[] longs, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
long	**getMappedAddress()**

	short	**getShort(long offset)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**getShorts(long offset, short[] shorts, int low, int number)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	long	**map()**
	long	**map(long base)**
	long	**map(long base, long size)**
❇		**RawMemoryAccess(Object type, long size)** *throws* **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**RawMemoryAccess(Object type, long base, long size)** *throws* **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
	void	**setByte(long offset, byte value)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setBytes(long offset, byte[] bytes, int low, int number)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setInt(long offset, int value)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setInts(long offset, int[] ints, int low, int number)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setLong(long offset, long value)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setLongs(long offset, long[] longs, int low, int number)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setShort(long offset, short value)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setShorts(long offset, short[] shorts, int low, int number)** *throws* **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**unmap()**

RawMemoryFloatAccess javax.realtime

Object
➥RawMemoryAccess
➥RawMemoryFloatAccess

	double	**getDouble(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**getDoubles(long offset, double[] doubles, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	float	**getFloat(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**getFloats(long offset, float[] floats, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
❇		**RawMemoryFloatAccess(Object type, long size)** ***throws*** **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**RawMemoryFloatAccess(Object type, long base, long size)** ***throws*** **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
	void	**setDouble(long offset, double value)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setDoubles(long offset, double[] doubles, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setFloat(long offset, float value)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**
	void	**setFloats(long offset, float[] floats, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException**

RealtimeSecurity javax.realtime

Object
➥RealtimeSecurity

	void	**checkAccessPhysical()** ***throws*** **SecurityException**
	void	**checkAccessPhysicalRange(long base, long size)** ***throws*** **SecurityException**
	void	**checkSetFilter()** ***throws*** **SecurityException**
	void	**checkSetScheduler()** ***throws*** **SecurityException**
❇		**RealtimeSecurity()**

RealtimeSystem javax.realtime

Object
➥RealtimeSystem

✍■	byte	**BIG_ENDIAN**
✍■	byte	**BYTE_ORDER**
❏	GarbageCollector	**currentGC()**
❏	int	**getConcurrentLocksUsed()**
❏	int	**getMaximumConcurrentLocks()**
❏	RealtimeSecurity	**getSecurityManager()**
✍■	byte	**LITTLE_ENDIAN**
❇		**RealtimeSystem()**
❏	void	**setMaximumConcurrentLocks(int numLocks)**
❏	void	**setMaximumConcurrentLocks(int number, boolean hard)**
❏	void	**setSecurityManager(RealtimeSecurity manager)**

RealtimeThread javax.realtime

Object
➥Thread Runnable
➥RealtimeThread Schedulable

	boolean	**addIfFeasible()**
	boolean	**addToFeasibility()**
❏	RealtimeThread	**currentRealtimeThread()** ***throws*** **ClassCastException**
	void	**deschedulePeriodic()**
❏	MemoryArea	**getCurrentMemoryArea()**
❏	int	**getInitialMemoryAreaIndex()**
	MemoryArea	**getMemoryArea()**

❑	int	**getMemoryAreaStackDepth()**
	MemoryParameters	**getMemoryParameters()**
❑	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)**
	void	**setMemoryParameters(MemoryParameters parameters) *throws* IllegalThreadStateException**
	boolean	**setMemoryParametersIfFeasible(MemoryParameters memory)**
	void	**setProcessingGroupParameters(ProcessingGroupParameters parameters)**
	boolean	**setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)**
	void	**setReleaseParameters(ReleaseParameters release) *throws* IllegalThreadStateException**
	boolean	**setReleaseParametersIfFeasible(ReleaseParameters release)**
	void	**setScheduler(Scheduler scheduler) *throws* IllegalThreadStateException**

	void	**setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group) *throws* IllegalThreadStateException**
	void	**setSchedulingParameters(SchedulingParameters scheduling) *throws* IllegalThreadStateException**
	boolean	**setSchedulingParametersIfFeasible(SchedulingParameters scheduling)**
❏	void	**sleep(Clock clock, HighResolutionTime time) *throws* InterruptedException**
❏	void	**sleep(HighResolutionTime time) *throws* InterruptedException**
	void	**start()**
	boolean	**waitForNextPeriod() *throws* IllegalThreadStateException**

RelativeTime javax.realtime

Object
➥HighResolutionTime Comparable
➥RelativeTime

	AbsoluteTime	**absolute(Clock clock)**
	AbsoluteTime	**absolute(Clock clock, AbsoluteTime destination)**
	RelativeTime	**add(long millis, int nanos)**
	RelativeTime	**add(long millis, int nanos, RelativeTime destination)**
●	RelativeTime	**add(RelativeTime time)**
	RelativeTime	**add(RelativeTime time, RelativeTime destination)**
	void	**addInterarrivalTo(AbsoluteTime timeAndDestination)**
	RelativeTime	**getInterarrivalTime()**
	RelativeTime	**getInterarrivalTime(RelativeTime destination)**
	RelativeTime	**relative(Clock clock)**
	RelativeTime	**relative(Clock clock, RelativeTime destination)**
❇		**RelativeTime()**
❇		**RelativeTime(long millis, int nanos)**
❇		**RelativeTime(RelativeTime time)**
●	RelativeTime	**subtract(RelativeTime time)**
	RelativeTime	**subtract(RelativeTime time, RelativeTime destination)**
	String	**toString()**

ReleaseParameters — javax.realtime

```
Object
  ➥ReleaseParameters
```

	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

```
Object
  ➥Throwable                    java.io.Serializable
     ➥Error
        ➥ResourceLimitError
```

❇		**ResourceLimitError()**
❇		**ResourceLimitError(String description)**

Schedulable — javax.realtime

```
Schedulable                     Runnable
```

	boolean	**addIfFeasible()**
	boolean	**addToFeasibility()**
	MemoryParameters	**getMemoryParameters()**
	ProcessingGroupParameters	**getProcessingGroupParameters()**
	ReleaseParameters	**getReleaseParameters()**
	Scheduler	**getScheduler()**
	SchedulingParameters	**getSchedulingParameters()**
	boolean	**removeFromFeasibility()**

	boolean	**setIfFeasible(ReleaseParameters release, MemoryParameters memory)**
	boolean	**setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)**
	boolean	**setIfFeasible(ReleaseParameters release, ProcessingGroupParameters group)**
	void	**setMemoryParameters(MemoryParameters memory)**
	boolean	**setMemoryParametersIfFeasible(MemoryParameters memory)**
	void	**setProcessingGroupParameters(ProcessingGroupParameters group)**
	boolean	**setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)**
	void	**setReleaseParameters(ReleaseParameters release)**
	boolean	**setReleaseParametersIfFeasible(ReleaseParameters release)**
	void	**setScheduler(Scheduler scheduler)** ***throws*** **IllegalThreadStateException**
	void	**setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters group)** ***throws*** **IllegalThreadStateException**
	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)**

SchedulingParameters **javax.realtime**

Object
  ➥SchedulingParameters

❇	**SchedulingParameters()**

ScopedCycleException **javax.realtime**

Object
  ➥Throwable   java.io.Serializable
    ➥Exception
      ➥RuntimeException
        ➥ScopedCycleException

❇	**ScopedCycleException()**
❇	**ScopedCycleException(String description)**

ScopedMemory **javax.realtime**

Object
  ➥MemoryArea
    ➥ScopedMemory

void	**enter()** ***throws*** **ScopedCycleException**
void	**enter(Runnable logic)** ***throws*** **ScopedCycleException**
long	**getMaximumSize()**
Object	**getPortal()**
int	**getReferenceCount()**
void	**join()** ***throws*** **InterruptedException**
void	**join(HighResolutionTime time)** ***throws*** **InterruptedException**
void	**joinAndEnter()** ***throws*** **InterruptedException, Scoped-CycleException**
void	**joinAndEnter(HighResolutionTime time)** ***throws*** **InterruptedException, ScopedCycleEx-ception**

	void	**joinAndEnter(Runnable logic)** ***throws*** **InterruptedException, ScopedCycleException**
	void	**joinAndEnter(Runnable logic, HighResolutionTime time)** ***throws*** **InterruptedException, ScopedCycleException**
❇		**ScopedMemory(long size)**
❇		**ScopedMemory(long size, Runnable runnable)**
❇		**ScopedMemory(SizeEstimator size)**
❇		**ScopedMemory(SizeEstimator size, Runnable runnable)**
	void	**setPortal(Object object)**
	String	**toString()**

SizeEstimator — javax.realtime

Object
➥SizeEstimator

	long	**getEstimate()**
	void	**reserve(Class clas, int number)**
	void	**reserve(SizeEstimator size)**
	void	**reserve(SizeEstimator size, int number)**
❇		**SizeEstimator()**

SizeOutOfBoundsException — javax.realtime

Object
➥Throwable java.io.Serializable
➥Exception
➥SizeOutOfBoundsException

❇		**SizeOutOfBoundsException()**
❇		**SizeOutOfBoundsException(String description)**

SporadicParameters — javax.realtime

Object
➥ReleaseParameters
➥AperiodicParameters
➥SporadicParameters

✍■	String	**arrivalTimeQueueOverflowExcept**
✍■	String	**arrivalTimeQueueOverflowIgnore**
✍■	String	**arrivalTimeQueueOverflowReplace**

✍■	String	**arrivalTimeQueueOverflowSave**
	String	**getArrivalTimeQueueOverflowBehavior()**
	int	**getInitialArrivalTimeQueueLength()**
	RelativeTime	**getMinimumInterarrival()**
	String	**getMitViolationBehavior()**
✍■	String	**mitViolationExcept**
✍■	String	**mitViolationIgnore**
✍■	String	**mitViolationReplace**
✍■	String	**mitViolationSave**
	void	**setArrivalTimeQueueOverflowBehavior(String behavio r)**
	boolean	**setIfFeasible(RelativeTime interarrival, RelativeTime cost, RelativeTime deadline)**
	void	**setInitialArrivalTimeQueueLength(int initial)**
	void	**setMinimumInterarrival(RelativeTime minimum)**
	void	**setMitViolationBehavior(String behavior)**
❊		**SporadicParameters(RelativeTime minInterarrival, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)**

ThrowBoundaryError **javax.realtime**

```
Object
  ➥Throwable                    java.io.Serializable
    ➥Error
      ➥ThrowBoundaryError
```

❊		**ThrowBoundaryError()**
❊		**ThrowBoundaryError(String description)**

Timed **javax.realtime**

```
Object
  ➥Throwable                    java.io.Serializable
    ➥Exception
      ➥InterruptedException
        ➥AsynchronouslyInterruptedException
          ➥Timed
```

	boolean	**doInterruptible(Interruptible logic)**
	void	**resetTime(HighResolutionTime time)**
❊		**Timed(HighResolutionTime time) *throws* IllegalArgumentException**

Timer — javax.realtime

```
Object
  ➥AsyncEvent
    ➥Timer
```

	ReleaseParameters	**createReleaseParameters()**
	void	**destroy()**
	void	**disable()**
	void	**enable()**
	Clock	**getClock()**
	AbsoluteTime	**getFireTime()**
	void	**reschedule(HighResolutionTime time)**
	void	**start()**
❇♦		**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
      ➥UnsupportedPhysicalMemoryException
```

❇		**UnsupportedPhysicalMemoryException()**
❇		**UnsupportedPhysicalMemoryException(String description)**

VTMemory javax.realtime

Object
➥MemoryArea
➥ScopedMemory
➥VTMemory

	long	**getMaximumSize()**
	String	**toString()**
❇		**VTMemory(long initial, long maximum)**
❇		**VTMemory(long initial, long maximum, 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) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**VTPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**VTPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**VTPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**

❇		**VTPhysicalMemory(Object type, long base, SizeEstimator size)** ***throws*** **SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**VTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
❇		**VTPhysicalMemory(Object type, SizeEstimator size)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
❇		**VTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)** ***throws*** **SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**

WaitFreeDequeue javax.realtime

Object
➥WaitFreeDequeue

	Object	**blockingRead()**
	boolean	**blockingWrite(Object object)** ***throws*** **MemoryScopeException**
	boolean	**force(Object object)**
	Object	**nonBlockingRead()**
	boolean	**nonBlockingWrite(Object object)** ***throws*** **MemoryScopeException**
❇		**WaitFreeDequeue(Thread writer, Thread reader, int maximum, MemoryArea area)** ***throws*** **IllegalArgumentException**

WaitFreeReadQueue javax.realtime

Object
➥WaitFreeReadQueue

	void	**clear()**
	boolean	**isEmpty()**
	boolean	**isFull()**
	Object	**read()**
	int	**size()**

	void	**waitForData()**
❇		**WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory) *throws* IllegalArgumentException**
❇		**WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory, boolean notify) *throws* IllegalArgumentException**
	boolean	**write(Object object) *throws* MemoryScopeException**

WaitFreeWriteQueue javax.realtime

Object
➥WaitFreeWriteQueue

	void	**clear()**
	boolean	**force(Object object) *throws* MemoryScopeException**
	boolean	**isEmpty()**
	boolean	**isFull()**
	Object	**read()**
	int	**size()**
❇		**WaitFreeWriteQueue(Thread writer, Thread reader, int maximum, MemoryArea memory) *throws* IllegalArgumentException**
	boolean	**write(Object object) *throws* MemoryScopeException**

Bibliography

1. J.H. Anderson, S. Ramamurthy, and K. Jeffay, *Real-Time Computing with Lock-Free Shared Objects*, IEEE Real-Time Systems Symposium 1995, pp. 28-37.
2. J. Anderson, R. Jain, S. Ramamurthy, *Wait-Free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors*, IEEE Real-Time Systems Symposium 1997, pp. 111-122.
3. H. Attiya and N.A. Lynch, *Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty*, IEEE Real-Time Systems Symposium 1989, pp. 268-284.
4. T.P. Baker and A. Shaw, *The Cyclic Executive Model and Ada*, IEEE Real-Time Systems Symposium 1988, pp. 120-129.
5. T.P. Baker, *A Stack-Based Resource Allocation Policy for Realtime Processes*, IEEE Real-Time Systems Symposium 1990, pp. 191-200.
6. T. Baker and O. Pazy, *Real-Time Features for Ada 9X*, IEEE Real-Time Systems Symposium 1991, pp. 172-180.
7. S.K. Baruah, A.K. Mok, and L.E. Rosier, *Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor*, IEEE Real-Time Systems Symposium 1990, pp. 182-190.
8. L. Carnahan and M. Ruark (eds.), *Requirements for Real-Time Extensions for the Java Platform*, National Institute of Standards and Technology, September 1999. Available at http://www.nist.gov/rt-java.
9. P. Chan, R. Lee, and D. Kramer, *The Java Class Libraries, Second Edition, Volume 1, Supplement for the Java 2 Platform, Standard Edition, v1.2,* Addison-Wesley, 1999.
10. M.-Z. Chen and K.J. Lin, *A Priority Ceiling Protocol for Multiple-Instance Resources*, IEEE Real-Time Systems Symposium 1991, pp. 140-149.
11. S. Cheng, J.A. Stankovic, and K. Ramamritham, *Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems*, IEEE Real-Time Systems Symposium 1986, pp. 166-174.
12. R.I. Davis, K. W. Tindell, and A. Burns, *Scheduling Slack Time in Fixed Priority Preemptive Systems*, IEEE Real-Time Systems Symposium 1993, pp. 222-231.
13. B.O. Gallmeister and C. Lanier, *Early Experience with POSIX 1003.4 and POSIX 1003.4a*, IEEE Real-Time Systems Symposium 1991, pp. 190-198.
14. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-

Wesley, 1996.

15. M.L. Green, E.Y.S. Lee, S. Majumdar, D.C. Shannon, *A Distributed Real Time Operating System*, IEEE Real-Time Systems Symposium 1980, pp. 175-184.

16. M.G. Harbour, M.H. Klein, and J.P. Lehoczky, *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, IEEE Real-Time Systems Symposium 1991, pp. 116-128.

17. F. Jahanian and A.K. Mok, *A Graph-Theoretic Approach for Timing Analysis in Real Time Logic*, IEEE Real-Time Systems Symposium 1986, pp. 98-108.

18. K. Jeffay, *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, IEEE Real-Time Systems Symposium 1989, pp. 295-307.

19. K. Jeffay, D.F. Stanat, and C.U. Martel , *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks*, IEEE Real-Time Systems Symposium 1991, pp. 129-139.

20. K. Jeffay, *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, IEEE Real-Time Systems Symposium 1992, pp. 89-99.

21. K. Jeffay and D.L. Stone, *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*, IEEE Real-Time Systems Symposium 1993, pp. 212-221.

22. K. Jeffay and D. Bennett, *A Rate-Based Execution Abstraction for Multimedia Computing*, Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (Apr. 1995).

23. E.D. Jensen, C.D. Locke, and H. Tokuda, *A Time-Driven Scheduling Model for Real-Time Operating Systems*, IEEE Real-Time Systems Symposium 1985, pp. 112-133.

24. Mark S. Johnstone, *Non-Compacting Memory Allocation and Real-Time Garbage Collection*, Ph.D. dissertation, The University of Texas at Austin, December 1997.

25. M.B. Jones, *Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services*, Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (Nov. 1993).

26. M.B. Jones, P.J. Leach, R.P. Draves, and J.S. Barrera, *Support for User-centric Modular Real-Time Resource Management in the Rialto Operating System*, Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (Apr. 1995).

27. I. Lee and S.B. Davidson, *Protocols for Timed Synchronous Process Communications*, IEEE Real-Time Systems Symposium 1986, pp. 120-137.

28. J.P. Lehoczky, L. Sha, and J.K. Strosnider, *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*, IEEE Real-Time Systems Symposium 1987, pp. 261-270.
29. J. Lehoczky, L. Sha, and Y. Ding, *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*, IEEE Real-Time Systems Symposium 1989, pp. 166-171.
30. J.P. Lehoczky and T.P. Baker, *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*, IEEE Real-Time Systems Symposium 1990, pp. 201-213.
31. J.P. Lehoczky and S. Ramos-Thuel, *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive System*, IEEE Real-Time Systems Symposium 1992, pp. 110-124.
32. K.-J. Lin, S. Natarajan, and J.W.-S. Liu, *Imprecise Results: Utilizing Partial Computations in Real-Time Systems*, IEEE Real-Time Systems Symposium 1987, pp. 210-218.
33. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition,* Addison-Wesley, 1999.
34. C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, JACM 20, 1 (Jan. 1973), pp. 46-61.
35. J.W.-S. Liu, K.-J. Lin, and S. Natarajan, *Scheduling Real-Time, Periodic Jobs Using Imprecise Results*, IEEE Real-Time Systems Symposium 1987, pp. 252-260.
36. C. Lizzi, *Enabling Deadline Scheduling for Java Real-Time Computing*, IEEE Real-Time Systems Symposium 1999.
37. C.D. Locke, D.R. Vogel, and T.J. Mesler, *Building a Predictable Avionics Platform in Ada: A Case Study*, IEEE Real-Time Systems Symposium 1991, pp. 180-189.
38. N. Lynch and N. Shavit, *Timing-Based Mutual Exclusion*, IEEE Real-Time Systems Symposium 1992, pp. 2-11.
39. C.W. Mercer and H. Tokuda, *Preemptibility in Real-Time Operating Systems*, IEEE Real-Time Systems Symposium 1992, pp. 78-88.
40. C.W. Mercer, S. Savage, and H. Tokuda, *Processor Capacity Reserves for Multimedia Operating Systems*, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (May 1994).
41. A. Miyoshi, T. Kitayama, H. Tokuda, *Implementation and Evaluation of Real-Time Java Threads*, IEEE Real-Time Systems Symposium 1997, pp. 166-175.
42. J.S. Ostroff and W.M. Wonham, *Modelling, Specifying and Verifying Real-Time Embedded Computer Systems*, IEEE Real-Time Systems Symposium 1987, pp.

124-132.

43. *Portable Operating System Interface (POSIX®) Part 1: System Application Program Interface*, International Standard ISO/IEC 9945-1: 1996 (E) IEEE Std 1003.1, 1996 Edition, The Institute of Electrical and Electronics Engineers, Inc. 1996.
44. R. Rajkumar, L. Sha, and J.P. Lehoczky, *On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment*, IEEE Real-Time Systems Symposium 1987, pp. 2-11.
45. R. Rajkumar, L. Sha, and J.P. Lehoczky, *Real-Time Synchronization Protocols for Multiprocessors*, IEEE Real-Time Systems Symposium 1988, pp. 259-271.
46. S. Ramos-Thuel and J.P. Lehoczky, *On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems*, IEEE Real-Time Systems Symposium 1993, pp. 160-171.
47. L. Sha, J.P. Lehoczky, and R. Rajkumar, *Solutions for Some Practical Problems in Prioritized Preemptive Scheduling*, IEEE Real-Time Systems Symposium 1986, pp. 181-193.
48. L. Sha, R. Rajkumar, and J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, Sept., 1990.
49. L. Sha, R. Rajkumar, and J. Lehoczky, *Real-Time Computing using Futurebus+*, IEEE Micro, June, 1991.
50. A.C. Shaw, *Software Clocks, Concurrent Programming, and Slice-Based Scheduling*, IEEE Real-Time Systems Symposium 1986, pp. 14-19.
51. F. Siebert, *Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor*, IEEE Real-Time Systems Symposium 1999.
52. B. Sprunt, J. Lehoczky, and L. Sha, *Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm*, IEEE Real-Time Systems Symposium 1988, pp. 251-258.
53. Sun Microsystems, Inc., *The Java Community Process Manual*, December 1998, Available at http://java.sun.com/aboutJava/communityprocess/java_community_process.html.
54. S.R. Thuel and J.P. Lehoczky, *Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems Using Slack Stealing*, IEEE Real-Time Systems Symposium 1994, pp. 22-35.
55. H. Tokuda, J.W. Wendorf, and H.-Y. Wang, *Implementation of a Time-Driven Scheduler for Real-Time Operating System*, IEEE Real-Time Systems Symposium 1987, pp. 271-280.

56. D.M. Washabaugh and D. Kafura, *Incremental Garbage Collection of Concurrent Objects for Real-Time Applications*, IEEE Real-Time Systems Symposium 1990, pp. 21-31.
57. P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review*, In International Workshop on Memory Management, Kinross, Scotland, UK, September 1995.
58. W. Zhao and K. Ramamritham, *A Virtual Time CSMA Protocol for Hard Real Time Communication*, IEEE Real-Time Systems Symposium 1986, pp. 120-127.
59. W. Zhao and J.A. Stankovic, *Performance Analysis of FCFS and Improved FCFS Scheduling Algorithms for Dynamic Real-Time Computer Systems*, IEEE Real-Time Systems Symposium 1989, pp. 156-165.

Colophon

This specification document was generated from a set of Java and HTML source files. They were compiled using `javadoc` and the doclet-from-hell: `mifdoclet.` The recent development of `mifdoclet` was driven largely by the Real Time for Java Expert Group. We wanted to be able to produce a specification document that had been checked, as much as possible, by whatever compilation tools we could find. The specification source compiles as a Java program, and even contains a scaffold implementation which was used to compile and run the examples.

The `mifdoclet` generates is output in MIF format, which was processed through Adobe FrameMaker, http://www.adobe.com/products/framemaker, a truely wonderful publishing package without which this book would have been much more difficult.

The source files used to produce this specification will eventually be available at http://www.rtj.org

Index