The Real-Time Specification for Java™

The Real-Time for Java Expert Group
http://www.rtj.org
Greg Bollella
Ben Bros gol 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
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.
Obsolete current version is 1.0.1
Contents

1 Introduction ..................................................................................... 1
2 Design ............................................................................................... 5
3 Threads .......................................................................................... 21
   RealtimeThread .................................................................................. 23
   NoHeapRealtimeThread ..................................................................... 33
4 Scheduling ...................................................................................... 37
   Schedulable ......................................................................................... 41
   Scheduler ............................................................................................ 45
   PriorityScheduler ................................................................................ 47
   SchedulingParameters ........................................................................ 51
   PriorityParameters .............................................................................. 51
   ImportanceParameters ......................................................................... 52
   ReleaseParameters ............................................................................... 54
   PeriodicParameters ........................................................................... 57
   AperiodicParameters ........................................................................... 59
   SporadicParameters ............................................................................ 61
   ProcessingGroupParameters ............................................................... 67
5 Memory Management ..................................................................... 71
   MemoryArea ....................................................................................... 77
   HeapMemory ...................................................................................... 81
   ImmortalMemory ............................................................................... 82
   SizeEstimator ..................................................................................... 82
   ScopedMemory ................................................................................... 84
   VTMemory ......................................................................................... 90
   LTMemory .......................................................................................... 92
   PhysicalMemoryManager ................................................................... 95
   PhysicalMemoryTypeFilter ................................................................. 98
   ImmortalPhysicalMemory ................................................................. 100
   LTPhysicalMemory .......................................................................... 106
   VTPhysicalMemory ......................................................................... 112
   RawMemoryAccess ......................................................................... 117
   RawMemoryFloatAccess ....................................................................... 125
   MemoryParameters ............................................................................ 129
   GarbageCollector .............................................................................. 132
6 Synchronization ............................................................................. 135
<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
<tr>
<td>MonitorControl .......................................................... 136</td>
</tr>
<tr>
<td>PriorityCeilingEmulation ................................. 138</td>
</tr>
<tr>
<td>PriorityInheritance .............................................. 138</td>
</tr>
<tr>
<td>WaitFreeWriteQueue ................................................. 139</td>
</tr>
<tr>
<td>WaitFreeReadQueue .................................................. 141</td>
</tr>
<tr>
<td>WaitFreeDequeue ...................................................... 144</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>Time ........................................................................ 147</td>
</tr>
<tr>
<td>HighResolutionTime .................................................. 148</td>
</tr>
<tr>
<td>AbsoluteTime ............................................................ 152</td>
</tr>
<tr>
<td>RelativeTime ............................................................. 156</td>
</tr>
<tr>
<td>RationalTime ............................................................. 160</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>Timers ....................................................................... 165</td>
</tr>
<tr>
<td>Clock ................................................................. 166</td>
</tr>
<tr>
<td>Timer .............................................................. 168</td>
</tr>
<tr>
<td>OneShotTimer ............................................................ 170</td>
</tr>
<tr>
<td>PeriodicTimer .......................................................... 171</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>Asynchrony ............................................................... 175</td>
</tr>
<tr>
<td>AsyncEvent .............................................................. 181</td>
</tr>
<tr>
<td>AsyncEventHandler ....................................................... 183</td>
</tr>
<tr>
<td>BoundAsyncEventHandler ............................................. 195</td>
</tr>
<tr>
<td>Interruptible ............................................................. 197</td>
</tr>
<tr>
<td>AsynchronouslyInterruptedException ..................... 198</td>
</tr>
<tr>
<td>Timed ..................................................................... 201</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>System and Options .................................................. 203</td>
</tr>
<tr>
<td>POSIXSignalHandler ................................................... 204</td>
</tr>
<tr>
<td>RealtimeSecurity ......................................................... 209</td>
</tr>
<tr>
<td>RealtimeSystem .......................................................... 210</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>Exceptions ................................................................. 213</td>
</tr>
<tr>
<td>DuplicateFilterException ........................................ 214</td>
</tr>
<tr>
<td>InaccessibleAreaException ........................................ 214</td>
</tr>
<tr>
<td>MemoryTypeConflictException .................................... 215</td>
</tr>
<tr>
<td>MemoryScopeException ................................................. 216</td>
</tr>
<tr>
<td>MITViolationException ............................................... 216</td>
</tr>
<tr>
<td>OffsetOutOfBoundsException ....................................... 217</td>
</tr>
<tr>
<td>SizeOutOfBoundsException .......................................... 217</td>
</tr>
<tr>
<td>UnsupportedPhysicalMemoryException .................... 218</td>
</tr>
<tr>
<td>MemoryInUseException ............................................... 219</td>
</tr>
<tr>
<td>ScopedCycleException ............................................... 219</td>
</tr>
<tr>
<td>UnknownHappeningException ..................................... 220</td>
</tr>
<tr>
<td>IllegalAssignmentError ............................................... 220</td>
</tr>
<tr>
<td>MemoryAccessError ..................................................... 221</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

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.
CHAPTER 1 INTRODUCTION

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

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

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

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

**Asynchronous Event Handling:** Real-time systems typically interact closely with the real-world. With respect to the execution of logic, the real-world is asynchronous. We thus felt compelled to include efficient mechanisms for programming disciplines that would accommodate this inherent asynchrony. The RTSJ generalizes the Java language’s mechanism of asynchronous event handling. Required classes represent things that can happen and logic that executes when those things happen. A notable feature is that the execution of the logic is scheduled and dispatched by an implemented scheduler.
Asynchronous Transfer of Control: Sometimes the real-world changes so drastically (and asynchronously) that the current point of logic execution should be immediately and efficiently transferred to another location. The RTSJ includes a mechanism which extends Java’s exception handling to allow applications to programatically change the locus of control of another Java thread. It is important to note that the RTSJ restricts this asynchronous transfer of control to logic specifically written with the assumption that its locus of control may asynchronously change.

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

Physical Memory Access: Although not directly a real-time issue, physical memory access is desirable for many of the applications that could productively make use of an implementation of the RTSJ. We thus define a class that allows programmers byte-level access to physical memory as well as a class that allows the construction of objects in physical memory.
Chapter 2

Design

The RTSJ comprises eight 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 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, 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 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).

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 times (thread create-time or set (reset)), 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 realizing 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 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).
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 stating exactly the algorithm used for granting such access.

Priority Inversion Avoidance

Any conforming implementation must provide an implementation of the synchronized primitive with default behavior that ensures that there is no unbounded priority inversion. Furthermore, this must apply to code if it is run within the implementation as well as to real-time threads. The priority inheritance protocol must be implemented by default. The priority inheritance protocol is a well-known algorithm in the real-time scheduling literature and it has the following effect. If
thread $t_1$ attempts to acquire a lock that is held by a lower-priority thread $t_2$, then $t_2$’s priority is raised to that of $t_1$ as long as $t_2$ holds the lock (and recursively if $t_2$ is itself waiting to acquire a lock held by an even lower-priority thread).

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

A second policy, priority ceiling emulation protocol (or highest locker protocol), is also specified for systems that support it. 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` have 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.

One needs also to consider the possible issues that could arise when a NoHeapRealtimeThread and a RealtimeThread attempt to synchronize on the same object. In this case if the NoHeapRealtimeThread blocks on the synchronization with the RealtimeThread and the RealtimeThread gets into a situation where the garbage collector will run, then the NoHeapRealtimeThread will find itself blocked on the garbage collector due to normal boosting. In general, the synchronization with a thread that can do garbage collection is a situation to be avoided, or the programmer must be ready for the consequences.

**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.

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. 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, 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 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.

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 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 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. 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. No semantics other than the `set<type>()` and `get<type>()` 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 `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`.

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

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

**Physical Memory Areas**

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

**Exceptions**

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

The new exceptions introduced are:

- **AsynchronouslyInterruptedException**: Generated when a thread is asynchronously interrupted.
- **DuplicateFilterException**: `PhysicalMemoryManager` 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.
- **InaccessibleAreaException**: Thrown when an attempt is made to execute or allo-
cate from an allocation context that is not accessible on the scope stack of the current thread.

- **MITViolationException**: Thrown by the `fire()` method of an instance of `AsyncEvent` when the bound instance of `AsyncEventHandler` with a `ReleaseParameter` type of `SporadicParameters` has `mitViolationExcept` behavior and the minimum interarrival time gets violated.

- **MemoryScopeException**: Thrown by the wait-free queue implementation when an object is passed that is not compatible with both ends of the queue.

- **MemoryTypeConflictException**: Thrown when the `PhysicalMemoryManager` is given conflicting specification for memory. The conflict can be between two types in an array of memory type specifiers, or when the specified base address does not fall in the requested memory type.

- **OffsetOutOfBoundsException**: Generated by the physical memory classes when the given offset is out of bounds.

- **SizeOutOfBoundsException**: Generated by the physical memory classes when the given size is out of bounds.

### Runtime Exceptions

- **UnsupportedPhysicalMemoryException**: Generated by the physical memory classes when the requested physical memory is unsupported.

- **MemoryInUseException**: Thrown when an attempt is made to allocate a range of physical or virtual memory that is already in use.

- **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).

- **UnknownHappeningException**: Thrown when `bindTo()` is called with an illegal happening.

### Errors

- **IllegalAssignmentError**: Thrown on an attempt to make an illegal assignment.

- **MemoryAccessException**: Thrown by the JVM when a thread attempts to access memory that is not in scope.

- **ResourceLimitError**: Thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks.

- **ThrowBoundaryError**: A throwable tried to propagate into a scope where it was not accessible.
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 (t1,...,t28) with priorities (p1,...,p28), respectively, where the value of p1 was the highest priority and the value of p2 the next highest priority, etc., then for all executions of threads t1 through t28 thread t1 would always execute in preference to threads t2,...,t28 and thread t2 would always execute in preference to threads t3,...,t28, 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 `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.)
DESIGN

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.

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

\textbf{Caution:} \texttt{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 \texttt{Comparable} interface is available in Java'2 v1.2 and 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 \texttt{Comparable} interface.

2. The class \texttt{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 \texttt{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:
Explicit exceptions to these semantics may also be global at the Chapter, Class, or Method level.

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>null</td>
</tr>
<tr>
<td>type[]</td>
<td>null</td>
</tr>
<tr>
<td>String</td>
<td>Null</td>
</tr>
<tr>
<td>int</td>
<td>less than zero</td>
</tr>
<tr>
<td>long</td>
<td>less than zero</td>
</tr>
<tr>
<td>float</td>
<td>less than zero</td>
</tr>
<tr>
<td>boolean</td>
<td>N/A</td>
</tr>
<tr>
<td>Class</td>
<td>null</td>
</tr>
</tbody>
</table>
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.
CHAPTER 3 THREADS

1. The default scheduling policy must manage the execution of instances of Object that 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.

6. In the specific case in which an instance of NoHeapRealtimeThread and an instance of either RealtimeThread or Thread synchronize on the same object the following exception to the immediately previous statement applies. Although by virtue of either the default priority inheritance algorithm or other priority inversion avoidance algorithm the temporary execution priority of either the instance of RealtimeThread or Thread may be raised to that of the instance of NoHeapRealtimeThread this temporary execution priority will not cause the instance of RealtimeThread or Thread to execute in preference of or to interrupt any garbage collector. This exception has the effect of causing an instance of NoHeapRealtimeThread to wait for the garbage collector. However, two observations should be noted. Since the instance NoHeapRealtimeThread is synchronizing with a thread that may be blocked by the execution of the garbage collector it should expect to be blocked as well. The alternative, allowing an instance of either RealtimeThread or Thread to preempt the garbage collector, can easily cause a complete system failure.

7. Instances of the RealtimeThread class may have an execution eligibility logically lower than the garbage collector.

8. Changing values in SchedulingParameters, ProcessingParameters, ReleaseParameters, ProcessingGroupParameters, or use of Thread.setPriority() must not affect the correctness of any implemented priority inversion avoidance algorithm.

9. 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.

10. 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.

11. The default parameter values for an object implementing 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.

12. Instances of objects implementing the interface `Schedulable` can be placed in memory represented by instances of `ImmortalMemory`, `HeapMemory`, `LTPhysicalMemory`, `VTPhysicalMemory`, or `ImmortalPhysicalMemory`.

**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.

### 3.1 RealtimeThread

**Declaration**

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

**All Implemented Interfaces:** `java.lang.Runnabe, Schedulable`

**Direct Known Subclasses:** `NoHeapRealtimeThread`
CHAPTER 3 THREADS

Description:
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 type of PeriodicParameters, and 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.

3.1.1 Constructors

public RealtimeThread()
Create a real-time thread. All parameter values are null.

public RealtimeThread(SchedulingParameters scheduling)
Create a real-time thread with the given SchedulingParameters.
Parameters:
  scheduling - The SchedulingParameters associated with this
  (and possibly other RealtimeThread).

public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters release)
Create a real-time thread with the given SchedulingParameters and
ReleaseParameters.
Parameters:
  scheduling - The SchedulingParameters associated with this
  (and possibly other instances of RealtimeThread).
  release - The ReleaseParameters associated with this (and
            possibly other instances of RealtimeThread).

public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters release,
                      MemoryParameters memory,
                      MemoryArea area,
Create a real-time thread with the given characteristics and a `java.lang.Runnable`.

**Parameters:**
- `scheduling` - The `SchedulingParameters` associated with this (and possibly other instances of `RealtimeThread`).
- `release` - The `ReleaseParameters` associated with this (and possibly other instances of `RealtimeThread`).
- `memory` - The `MemoryParameters` associated with this (and possibly other instances of `RealtimeThread`).
- `area` - The `MemoryArea` associated with this.
- `group` - The `ProcessingGroupParameters` associated with this (and possibly other instances of `RealtimeThread`).

### 3.1.2 Methods

**public boolean addIfFeasible()**

Add to the feasibility of the already set scheduler if the resulting feasibility set is schedulable. If successful return true, if not return false. If there is not an assigned scheduler it will return false.

**public boolean addToFeasibility()**

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters`) of this instance of `Schedulable` will be considered in the feasibility analysis of the associated `Scheduler` until further notice. Whether the resulting system is feasible or not, the addition is completed.

**Specified By:** `public boolean addToFeasibility()` in interface `Schedulable`

**Returns:** true If the resulting system is feasible.

**public static RealtimeThread currentRealtimeThread()**

Throws `ClassCastException`
This will throw a `ClassCastException` if the current thread is not a `RealtimeThread`.

**Throws:**

`ClassCastException`

```java
public void deschedulePeriodic()
```

Stop unblocking a periodic schedulable object. If this does not have a type of `PeriodicParameters` as it `ReleaseParameters` nothing happens.

```java
public static MemoryArea getCurrentMemoryArea()
```

Return the instance of `MemoryArea` which is the current memory area for this.

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

Memory area stacks include inherited stacks from parent threads. The initial 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 initial memory area of the current `RealtimeThread`.

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

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

**Returns:** The size of the stack of `MemoryArea` instances.

```java
public MemoryParameters getMemoryParameters()
```

Return a reference to the `MemoryParameters` object.

**Specified By:** public `MemoryParameters getMemoryParameters()` in interface `Schedulable`

```java
public static MemoryArea getOuterMemoryArea(int index)
```


Get the instance of MemoryArea 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 at index or null if the given value is does not correspond to a position in the stack.

```java
public ProcessingGroupParameters getProcessingGroupParameters()
```

Return a reference to the ProcessingGroupParameters object.

**Specified By:** public ProcessingGroupParameters getProcessingGroupParameters() in interface Schedulable

```java
public ReleaseParameters getReleaseParameters()
```

Returns a reference to the ReleaseParameters object.

**Specified By:** public ReleaseParameters getReleaseParameters() in interface Schedulable

```java
public Scheduler getScheduler()
```

Get the scheduler for this thread.

**Specified By:** public Scheduler getScheduler() in interface Schedulable

```java
public SchedulingParameters getSchedulingParameters()
```

Return a reference to the SchedulingParameters object.

**Specified By:** public SchedulingParameters getSchedulingParameters() in interface Schedulable

```java
public void interrupt()
```

Throw the generic AsynchronouslyInterruptedException at this.

**Overrides:** java.lang.Thread.interrupt() in class java.lang.Thread

```java
public boolean removeFromFeasibility()
```

27
Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters`, of this instance of `Schedulable` should no longer be considered in the feasibility analysis of the associated `Scheduler`. Whether the resulting system is feasible or not, the subtraction is completed.

**Specified By:** public boolean `removeFromFeasibility()` in interface `Schedulable`

**Returns:** true if the resulting system is feasible.

```java
public void schedulePeriodic()
```

Begin unblocking public boolean `waitForNextPeriod()` throws `IllegalThreadStateException` 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` as it `ReleaseParameters` nothing happens.

```java
public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

```java
public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

```java
public boolean setIfFeasible(ReleaseParameters release, ProcessingGroupParameters group)
```
Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

```java
public void setMemoryParameters(MemoryParameters parameters)
    throws IllegalThreadStateException

Set the reference to the MemoryParameters object.

Specified By: public void
    setMemoryParameters(MemoryParameters memory)
in interface Schedulable

Throws:
    IllegalThreadStateException
```

```java
public boolean setMemoryParametersIfFeasible(MemoryParameters memParam)

Returns true if, after considering the value of the parameter, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the value of the parameter, the task set would not be feasible. In this case the values of the parameters are not changed.

Specified By: public boolean
    setMemoryParametersIfFeasible(MemoryParameters memParam)
in interface Schedulable
```

```java
public void setProcessingGroupParameters(ProcessingGroupParameters parameters)

Set the reference to the ProcessingGroupParameters object.

Specified By: public void
    setProcessingGroupParameters(ProcessingGroupParameters groupParameters)
in interface Schedulable
```

```java
public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)

```

```
```
CHAPTER 3 THREADS

Returns true if, after considering the value of the parameter, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the value of the parameter, the task set would not be feasible. In this case the values of the parameters are not changed.

Specified By: public boolean

setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)

In interface Schedulable

public void setReleaseParameters(ReleaseParameters parameters)

throws IllegalThreadStateException

Set the reference to the ReleaseParameters object.

Specified By: public void

setReleaseParameters(ReleaseParameters release)

in interface Schedulable

Throws:

IllegalThreadStateException

public boolean

setReleaseParametersIfFeasible(ReleaseParameters release)

Returns true if, after considering the value of the parameter, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the value of the parameter, the task set would not be feasible. In this case the values of the parameters are not changed.

Specified By: public boolean

setReleaseParametersIfFeasible(ReleaseParameters release)

in interface Schedulable

public void setScheduler(Scheduler scheduler)

throws IllegalThreadStateException

Set the scheduler. This is a reference to the scheduler that will manage the execution of this thread.
REALTIME_THREAD

Specified By: public void setScheduler(Scheduler scheduler) throws IllegalThreadStateException
Specified By: public void setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup) throws IllegalThreadStateException

Throws:

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

Set the scheduler. This is a reference to the scheduler that will manage the execution of this thread.

Specified By: public void setSchedulingParameters(SchedulingParameters scheduling)

Throws:

IllegalThreadStateException

Set the reference to the SchedulingParameters object.

Specified By: public void setSchedulingParameters(SchedulingParameters scheduling)

Throws:

IllegalThreadStateException
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)

Returns true if, after considering the value of the parameter, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the value of the parameter, the task set would not be feasible. In this case the values of the parameters are not changed.

Specified By: public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling) in interface Schedulable

public static void sleep(Clock clock, HighResolutionTime 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. 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.

Throws:
  InterruptedException

public static void sleep(HighResolutionTime 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. 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.

Throws:
  InterruptedException

public void start()
Checks if the instance of `RealtimeThread` is startable and starts it if it is.

Overrides: `java.lang.Thread.start()` in class `java.lang.Thread`

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

Used by threads that have a reference to a `ReleaseParameters` type of `PeriodicParameters` to block until the start of each period. Periods start at either the start time in `PeriodicParameters` 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, `public boolean waitForNextPeriod()` throws `IllegalThreadStateException` 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, `public boolean waitForNextPeriod()` throws `IllegalThreadStateException` will block until the handler corrects the situation (possibly by calling `public void schedulePeriodic()`).

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

Throws:
- `IllegalThreadStateException`

### 3.2 NoHeapRealtimeThread

**Declaration**

```java
public class NoHeapRealtimeThread extends RealtimeThread
```

**All Implemented Interfaces:** `java.langRunnable, Schedulable`  

**Description**

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`. 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 is it even allowed to manipulate any reference to
any object 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 anything 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 or ImmortalMemory.

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.

3.2.1 Constructors

    public NoHeapRealtimeThread(SchedulingParameters sp,
                                  MemoryArea ma)
                                  throws IllegalArgumentException

Create a NoHeapRealtimeThread.

Parameters:

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

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

Throws:

    IllegalArgumentException

    public NoHeapRealtimeThread(SchedulingParameters sp,
                                  ReleaseParameters rp, MemoryArea ma)
                                  throws IllegalArgumentException
Create a `NoHeapRealtimeThread`.

**Parameters:**

- `scheduling` - A `SchedulingParameters` object that will be associated with this. A null value means this will not have an associated `SchedulingParameters` object.
- `release` - A `ReleaseParameters` object that will be associated with this. A null value means this will not have an associated `ReleaseParameters` object.
- `area` - A `MemoryArea` object. Must be a `ScopedMemory` or `ImmortalMemory` type. A null value causes an `IllegalArgumentException` to be thrown.

**Throws:**

- `IllegalArgumentException`

```java
public NoHeapRealtimeThread(SchedulingParameters sp,
                            ReleaseParameters rp,
                            MemoryParameters mp, MemoryArea ma,
                            ProcessingGroupParameters group,
                            java.lang.Runnable logic)
    throws IllegalArgumentException
```

Create a `NoHeapRealtimeThread`.

**Parameters:**

- `scheduling` - A `SchedulingParameters` object that will be associated with this. A null value means this will not have an associated `SchedulingParameters` object.
- `release` - A `ReleaseParameters` object that will be associated with this. A null value means this will not have an associated `ReleaseParameters` object.
- `memory` - A `MemoryParameters` object that will be associated with this. A null value means this will not have a `MemoryParameters` object.
- `area` - A `MemoryArea` object. Must be a `ScopedMemory` or `ImmortalMemory` type. A null value causes an `IllegalArgumentException` to be thrown.
- `group` - A `ProcessingGroupParameters` object that will be associated with this. A null value means this will not have an associated `ProcessingGroupParameters` object.
- `logic` - A Runnable whose `run()` method will be executed for this.
CHAPTER 3 THREADS

Throws:
   IllegalArgumentException

3.2.2 Methods

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.

Overrides: public void start() in class RealtimeThread

Throws:
   MemoryAccessError - If any of the parameters or this is allocated on heap.
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 `RealtimeThread` 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 `SchedulingParameters` 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 `PriorityParameters` 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 thread 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 sched-
ulable 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 pri-
ority 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 implementa-
tion. 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.
CHAPTER 4 SCHEDULING


11. If instances of `RealtimeThread` or `NoHeapRealtimeThread` are constructed without a reference to a `SchedulingParameters` object, a `SchedulingParameters` 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 referred 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 `MITVioationException` 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, 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.

### 4.1 Schedulable

**Declaration**

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

**All Superinterfaces:** `java.lang.Runnable`

**All Known Implementing Classes:** `AsyncEventHandler`, `RealtimeThread`

**Description**

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

#### 4.1.1 Methods

```java
public boolean addToFeasibility()
```
Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters`) of this instance of `Schedulable` will be considered in the feasibility analysis of the associated `Scheduler` until further notice. Whether the resulting system is feasible or not, the addition is completed.

*Returns:* true If the resulting system is feasible.

```java
public MemoryParameters getMemoryParameters()
{ return MemoryParameters of this schedulable object. }
```

```java
public ProcessingGroupParameters getProcessingGroupParameters()
{ return ProcessingGroupParameters of this schedulable object. }
```

```java
public ReleaseParameters getReleaseParameters()
{ return ReleaseParameters of this schedulable object. }
```

```java
public Scheduler getScheduler()
{ return Scheduler for this schedulable object. }
```

```java
public SchedulingParameters getSchedulingParameters()
{ return SchedulingParameters of this schedulable object. }
```

```java
public boolean removeFromFeasibility()
{ return true If the resulting system is feasible. }
```

```java
public void setMemoryParameters(MemoryParameters memory)
{ inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters` of this instance of `Schedulable` should no longer be considered in the feasibility analysis of the associated `Scheduler`. Whether the resulting system is feasible or not, the subtraction is completed.

*Returns:* true If the resulting system is feasible. }
```

```java
public void setMemoryParameters(MemoryParameters memory)
{ inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters` of this instance of `Schedulable` should no longer be considered in the feasibility analysis of the associated `Scheduler`. Whether the resulting system is feasible or not, the subtraction is completed.

*Returns:* true If the resulting system is feasible. }
```
Set the $\text{MemoryParameters}$ of this schedulable object.

$\text{Parameters:}$

- memory - The $\text{MemoryParameters}$ object.

```java
public boolean setMemoryParametersIfFeasible(MemoryParameters memParam)
```

Set the $\text{MemoryParameters}$ of this schedulable object.

$\text{Parameters:}$

- memory - The $\text{MemoryParameters}$ object. If null nothing happens.

```java
public void setProcessingGroupParameters(ProcessingGroupParameters groupParameters)
```

Set the $\text{ProcessingGroupParameters}$ of this schedulable object.

$\text{Parameters:}$

- groupParameters - The $\text{ProcessingGroupParameters}$ object.

```java
public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)
```

Set the $\text{ProcessingGroupParameters}$ of this schedulable object only if the resulting task set is feasible.

$\text{Parameters:}$

- groupParameters - The $\text{ProcessingGroupParameters}$ object.

```java
public void setReleaseParameters(ReleaseParameters release)
```

Set the $\text{ReleaseParameters}$ for this schedulable object.

$\text{Parameters:}$

- release - The $\text{ReleaseParameters}$ object.

```java
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)
```
CHAPTER 4 SCHEDULING

Set the ReleaseParameters for this schedulable object only if the resulting task set is feasible.

Parameters:
release - The ReleaseParameters object. If null nothing happens.

```java
public void setScheduler(Scheduler scheduler) throws IllegalThreadStateException
```

Set the Scheduler for this schedulable object.

Parameters:
scheduler - The Scheduler object.

Throws:
IllegalThreadStateException

```java
public void setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup) throws IllegalThreadStateException
```

Set the Scheduler for this schedulable object.

Parameters:
scheduler - The Scheduler object.

Throws:
IllegalThreadStateException

```java
public void setSchedulingParameters(SchedulingParameters scheduling)
```

Set the SchedulingParameters of this schedulable object.

Parameters:
scheduling - The SchedulingParameters object.

```java
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)
```

Set the SchedulingParameters of this schedulable object only if the resulting task set is feasible.
Parameters:

- scheduling - The `SchedulingParameters` object. If null nothing happens.

### 4.2 Scheduler

**Declaration**

```java
public abstract class Scheduler
```

**Direct Known Subclasses**: `PriorityScheduler`

**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 `public static Scheduler getDefaultScheduler()` (e.g., `EDFScheduler`).

#### 4.2.1 Constructors

```java
protected Scheduler()
```

Constructor.

#### 4.2.2 Methods

```java
protected abstract boolean addToFeasibility(Schedulable schedulable)
```

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters`) of this instance of `Schedulable` will be considered in the feasibility analysis of the associated `Scheduler` until further notice. Whether the resulting system is feasible or not, the addition is completed.

**Returns**: true If the resulting system is feasible.
public abstract void fireSchedulable(Schedulable schedulable)

Trigger the execution of a schedulable object (like an AsyncEventHandler).

Parameters:
   schedulable - The schedulable object to make active.

public static Scheduler getDefaultScheduler()

Return a reference to the default scheduler.

public abstract java.lang.String getPolicyName()

Used to determine the policy of the Scheduler.

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

public abstract boolean isFeasible()

Returns true if and only if the system is able to satisfy the constraints expressed in the release parameters of the existing schedulable objects.

protected abstract boolean removeFromFeasibility(Schedulable schedulable)

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

Returns: true If the resulting system is feasible.

public static void setDefaultScheduler(Scheduler scheduler)

Set the default scheduler. This is the scheduler given to instances of RealTimeThread when they are constructed. The default scheduler is set to the required PriorityScheduler at startup.
Parameters:

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

```
public boolean setIfFeasible(Schedulable schedulable,
ReleaseParameters release,
MemoryParameters memory)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

```
public boolean setIfFeasible(Schedulable schedulable,
ReleaseParameters release,
MemoryParameters memory,
ProcessingGroupParameters group)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

### 4.3 PriorityScheduler

Declaration:

```
public class PriorityScheduler extends Scheduler
```

#### 4.3.1 Fields

```
public static final int MAX_PRIORITY
```

```
public static final int MIN_PRIORITY
```

#### 4.3.2 Constructors

```
protected PriorityScheduler()
```

Constructor for the required scheduler.
4.3.3 Methods

```java
protected boolean addToFeasibility(Schedulable schedulable)
```

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`, `ReleaseParameters`, `MemoryParameters`, and `ProcessingGroupParameters`, of this instance of `Schedulable` will be considered in the feasibility analysis of the associated `Scheduler` until further notice. Whether the resulting system is feasible or not, the addition is completed.

```java
Overrides: protected abstract boolean addToFeasibility(Schedulable schedulable) in class Scheduler
```

`Returns:` true if the resulting system is feasible.

```java
public void fireSchedulable(Schedulable schedulable)
```

Trigger the execution of a schedulable object (like an instance of `AsyncEventHandler`).

```java
Overrides: public abstract void fireSchedulable(Schedulable schedulable) in class Scheduler
```

`Parameters:`

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

```java
public int getMaxPriority()
```

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

```java
public static int getMaxPriority(java.lang.Thread 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 `Thread`. If null the maximum priority of the required `PriorityScheduler` is returned.
public int getMinPriority()

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

public static int getMinPriority(java.lang.Thread 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 Thread. If null the minimum priority of the required PriorityScheduler is returned.

public int getNormPriority()

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

public static int getNormPriority(java.lang.Thread 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.

public java.lang.String getPolicyName()

Used to determine the policy of the Scheduler.

Overrides: public abstract java.lang.String getPolicyName() in class Scheduler

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

public static PriorityScheduler instance()

Return a pointer to an instance of PriorityScheduler.

public boolean isFeasible()
Returns true if and only if the system is able to satisfy the constraints expressed in the release parameters of the existing schedulable objects.

**Overrides:** public abstract boolean isFeasible() in class Scheduler

protected boolean removeFromFeasibility(Schedulable schedulable)

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

**Overrides:** protected abstract boolean removeFromFeasibility(Schedulable schedulable) in class Scheduler

Returns: true if the resulting system is feasible.

public boolean setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

**Overrides:** public boolean setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory) in class Scheduler

public boolean setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters,
the task set would not be feasible. In this case the values of the parameters are not changed.

Overrides: public boolean setIfFeasible(Schedulable\_41 schedulable, ReleaseParameters\_54 release, MemoryParameters\_123 memory, ProcessingGroupParameters\_67 group)\_47 in class Scheduler\_45

4.4 SchedulingParameters

Declaration : public abstract class SchedulingParameters

Direct Known Subclasses: PriorityParameters\_51

Description : Subclasses of SchedulingParameters (PriorityParameters\_51, ImportanceParameters\_52, and any others defined for particular schedulers) provide the parameters to be used by the Scheduler. Changes to the values in a parameters object affects the scheduling behavior of all the Schedulable 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.

4.4.1 Constructors

public SchedulingParameters()

4.5 PriorityParameters

Declaration : public class PriorityParameters extends SchedulingParameters\_51

Direct Known Subclasses: ImportanceParameters\_52

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
CHAPTER 4 SCHEDULING

by this specification and represented by the class PriorityScheduler is such a scheduler.

4.5.1 Constructors

public PriorityParameters(int priority)
Create an instance of SchedulingParameters 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).

4.5.2 Methods

public int getPriority()
Get the priority.

public void setPriority(int priority)
throws IllegalArgumentException
Set the priority.
Parameters:
priority - The new value of priority.
Throws:
IllegalArgumentException - Thrown if the given priority value is less than the minimum priority of the scheduler of any of the associated threads or greater than the maximum priority of the scheduler of any of the associated threads.

public java.lang.String toString()
Overrides: java.lang.Object.toString() in class java.lang.Object

4.6 ImportanceParameters

Declaration:
public class ImportanceParameters extends PriorityParameters


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

4.6.1 Constructors

```java
public ImportanceParameters(int priority, int importance)
```
Create an instance of `ImportanceParameters`.

Parameters:
- `priority` - The priority assigned to a thread. This value is used in place of `java.lang.Thread.priority`.
- `importance` - The importance value assigned to a thread.

4.6.2 Methods

```java
public int getImportance()
```
Get the importance value.

```java
public void setImportance(int importance)
```
Set the importance.

```java
public java.lang.String toString()
```
Overrides: `public java.lang.String toString() in class PriorityParameters`
CHAPTER 4 SCHEDULING

4.7 ReleaseParameters

Declaration

```java
public class ReleaseParameters
```

Direct Known Subclasses: AperiodicParameters, PeriodicParameters

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.

4.7.1 Constructors

```java
protected ReleaseParameters()
```

```java
protected ReleaseParameters(RelativeTime cost,
RelativeTime deadline,
AsyncEventHandler overrunHandler,
AsyncEventHandler missHandler)
```

Subclasses use this constructor to create a ReleaseParameters type object.

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 \texttt{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 \texttt{waitForNextPeriod} returns false immediately and updates the start time for the next period.

missHandler - This handler is invoked if the \texttt{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.

4.7.2 Methods

\texttt{public RelativeTime getCost()}  
Get the cost value.

\texttt{public AsyncEventHandler getCostOverrunHandler()}  
Get the cost overrun handler.

\texttt{public RelativeTime getDeadline()}  
Get the deadline.

\texttt{public AsyncEventHandler getDeadlineMissHandler()}  
Get the deadline miss handler.

\texttt{public void setCost(RelativeTime cost)}  
Set 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 $\text{RelativeTime}(0,0)$ if null.

```java
public void setCostOverrunHandler(AsyncEventHandler handler)
```

Set the cost overrun handler.

**Parameters:**

- **handler** - This handler is invoked if an invocation of the schedulable object exceeds cost. Not required for minimum implementation. See comments in `setCost()`.

```java
public void setDeadline(RelativeTime deadline)
```

Set 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.

```java
public void setDeadlineMissHandler(AsyncEventHandler handler)
```

Set 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
PERIODIC PARAMETERS

deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate.

```java
public boolean setIfFeasible(RelativeTime cost, RelativeTime deadline)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

### 4.8 PeriodicParameters

**Declaration**

```java
public class PeriodicParameters extends ReleaseParameters
```

**Description**

This release parameter indicates that the `waitForNextPeriod()` method on the associated `Schedulable` 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.

#### 4.8.1 Constructors

```java
public PeriodicParameters(HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)
```

Create a `PeriodicParameters` object.
Parameters:

**start** - Time at which the first period begins. If a RelativeTime, 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 and it is before the schedulable time, start is equivalent to the schedulable time.

**period** - The period is the interval between successive unblocks of public boolean waitForNextPeriod() throws IllegalThreadStateException. 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.

4.8.2 Methods

```java
public RelativeTime getPeriod()
```
Get the period.

```java
public HighResolutionTime getStart()
```

Get the start time.

```java
public boolean setIfFeasible(RelativeTime period,
                              RelativeTime cost,
                              RelativeTime deadline)
```

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

```java
public void setPeriod(RelativeTime p)
```

Set the period.

**Parameters:**
- `period` - The period is the interval between successive unblocks of

```java
public boolean waitForNextPeriod()
```

Also used in the feasibility analysis and admission control algorithms.

```java
public void setStart(HighResolutionTime s)
```

Set the start time.

**Parameters:**
- `start` - Time at which the first period begins.

## 4.9 AperiodicParameters

**Declaration**

```java
public class AperiodicParameters extends ReleaseParameters
```

**Direct Known Subclasses:** `SporadicParameters`

**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 \texttt{AperiodicParameters} object affect the constructed object. If given to more than one constructor then changes to the values in the \texttt{AperiodicParameters} object affect all of the associated objects. Note that this is a one-to-many relationship and \textit{not} a many-to-many.

\textbf{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.

4.9.1 Constructors

\begin{verbatim}
public AperiodicParameters(RelativeTime\_156\_time\_156\_cost, RelativeTime\_156\_deadline, AsyncEventHandler\_183\_overrunHandler, AsyncEventHandler\_183\_missHandler)

Create an AperiodicParameters object.

Parameters:
\begin{description}
\item[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(0,0) if null.
\item[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(Long.MAX_VALUE,999999).
\item[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.
\item[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
\end{itemize}
\end{verbatim}
variable deadlines and invoke the miss handler as appropriate. If null, nothing happens on the miss deadline condition.

4.9.2 Methods

```java
public boolean setIfFeasible(RelativeTime cost, RelativeTime deadline)
```

Attempt to change the cost and deadline. The values will be changed if the resulting system is feasible. If the resulting system would not be feasible no changes are made.

**Overrides:** public boolean setIfFeasible(RelativeTime cost, RelativeTime deadline) in class ReleaseParameters

**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.

4.10 SporadicParameters

**Declaration:**

```java
public class SporadicParameters extends AperiodicParameters
```

**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.
CHAPTER 4 SCHEDULING

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` the arrival time is the time at which the `start()` is invoked. For other instances of `Schedulable` it may be required for the implementation to save the arrival times. For instances of `AsyncEventHandler` with a `ReleaseParameters` type of `SporadicParameters` the implementation must maintain a queue of monotonically increasing arrival times which correspond to the execution of the `fire()` method of the instance of `AsyncEvent` bound to the instance of `AsyncEventHandler`.

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.

4.10.1 Fields

```java
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`. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler` is bound then the arrival time is ignored.

```java
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.

```java
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.
**SporadicParameters**

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.

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

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

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

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

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

If an arrival time for any instance of Schedulable which has this as its instance of ReleaseParameters occurs at a time less then the minimum interarrival time defined here then, if necessary, the previous arrival time may be overwritten with the new arrival time.

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

If an arrival time for any instance of Schedulable which has this as its instance of ReleaseParameters 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.

4.10.2 Constructors

public SporadicParameters(RelativeTime minInterarrival, RelativeTime cost,
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.

## 4.10.3 Methods

```java
public java.lang.String
getArrivalQueueOverflowBehavior();
```
Get the behavior of the arrival time queue in the event of an overflow.

```java
public java.lang.String getArrivalTimeQueueOverflowBehavior()
// Get the behavior of the arrival time queue in the event of an overflow.
```

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

```java
public int getInitialArrivalTimeQueueLength()
// Get the initial number of elements the arrival time queue can hold.
```

Get the minimum interarrival time.

```java
public RelativeTime getMinimumInterarrival()
// Get the minimum interarrival time.
```

Get the arrival time queue behavior in the event of a minimum interarrival time violation.

```java
public java.lang.String getMitViolationBehavior()
// Get the arrival time queue behavior in the event of a minimum interarrival time violation.
```

Set 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.

```java
public void setArrivalTimeQueueOverflowBehavior(java.lang.String behavior)
// Set 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.
```

Set 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.
public boolean setIfFeasible(RelativeTime interarrival, RelativeTime cost, RelativeTime deadline)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

public void setInitialArrivalTimeQueueLength(int initial)

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

public void setInitialArrivalTimeQueueLength(int initial)

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

public void setMinimumInterarrival(RelativeTime 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.

public void setMitViolationBehavior(java.lang.String behavior)

Set 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.

public void setMitViolationBehavior(java.lang.String behavior)

Set 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.
4.11 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 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.

4.11.1 Constructors

public ProcessingGroupParameters(HighResolutionTime start, RelativeTime period,
RelativeTime cost, RelativeTime deadline,
AsyncEventHandler overrunHandler,
AsyncEventHandler 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().
CHAPTER 4 SCHEDULING

**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.

### 4.11.2 Methods

```java
public RelativeTime getCost()
Get the cost value.
```

```java
public AsyncEventHandler getCostOverrunHandler()
Get the cost overrun handler.
Returns: An AsyncEventHandler object that is cost overrun handler of this.
```

```java
public RelativeTime getDeadline()
Get the deadline value.
Returns: A RelativeTime object that represents the deadline of this.
```

```java
public AsyncEventHandler getDeadlineMissHandler()
Get the deadline missed handler.
Returns: An AsyncEventHandler object that is deadline miss handler of this.
```

```java
public RelativeTime getPeriod()
Get the period.
Returns: A RelativeTime object that represents the period of time of this.
```
PROCESSINGGROUPPARAMETERS

public HighResolutionTime\_148\ getStart\()\n
Get the start time.

*Returns:* A `HighResolutionTime` object that represents the start time of this.

public void setCost\(\text{RelativeTime} \_156\ cost\)\n
Set the cost value.

*Parameters:*

- `cost` - The schedulable objects with a reference to this receive cumulatively no more than cost time per period on implementations that can collect execution time per thread.

public void setCostOverrunHandler\(\text{AsyncEventHandler} \_183\ handler\)\n
Set the cost overrun handler.

*Parameters:*

- `handler` - 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.

public void setDeadline\(\text{RelativeTime} \_156\ deadline\)\n
Set the deadline value.

*Parameters:*

- `deadline` - The latest permissible completion time measured from the start of the current period. Not used in a minimum implementation. Other implementations may use this parameter to compute execution eligibility. The default value is the same as the period.

public void setDeadlineMissHandler\(\text{AsyncEventHandler} \_183\ handler\)\n
Set 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.
public boolean setIfFeasible(RelativeTime period, RelativeTime cost, RelativeTime deadline)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

public void setPeriod(RelativeTime period)

Set the period.

Parameters:
period - Interval used to enforce allocation of processing resources to the associated schedulable objects. Also used in the feasibility analysis and admission control algorithms.

public void setStart(HighResolutionTime start)

Set the start time.

Parameters:
start - Time at which the first period begins.
Chapter 5

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 \( n > 0 \), \( f(n) \leq Cn \) 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 reference-able 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:

<table>
<thead>
<tr>
<th>Reference to Heap</th>
<th>Reference to Immortal</th>
<th>Reference to Scoped</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Immortal</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Scoped</td>
<td>Yes</td>
<td>Yes, if same, outer, or shared scope</td>
</tr>
<tr>
<td>Local Variable</td>
<td>Yes</td>
<td>Yes, if same, outer, or shared scope</td>
</tr>
</tbody>
</table>

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.
CHAPTER 5 MEMORY MANAGEMENT

- 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 thread
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 thread
  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 area 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 possible. 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.

```plaintext
precondition: ma.parent is set to the correct parent (either a scoped memory area or the primordial area) or to noParent.

1. 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

```plaintext
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, 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 throw 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.

5.1 MemoryArea

Declaration
public abstract class MemoryArea

Direct Known Subclasses: HeapMemory81, ImmortalMemory82, ImmortalPhysicalMemory100, ScopedMemory84

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.

5.1.1 Constructors

protected MemoryArea(long sizeInBytes)
Parameters:
sizeInBytes - The size of MemoryArea to allocate, in bytes.

protected MemoryArea(long sizeInBytes, java.lang.Runnable logic)
Parameters:
sizeInBytes - The size of MemoryArea to allocate, in bytes.
CHAPTER 5 MEMORY MANAGEMENT

logic - The run() method of this object will be called whenever

public void enter()
throws ScopedCycleException is called.

protected MemoryArea(SizeEstimator size)

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

protected MemoryArea(SizeEstimator size, java.lang.Runnable logic)

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

public void enter()
throws ScopedCycleException is called.

5.1.2 Methods

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. 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 or
NoHeapRealtimeThread.

Throws:
IllegalArgument Exception - Thrown if no Runnable was passed
in the constructor.
ScopedCycleException - If entering this ScopedMemory
would violate the single parent rule.

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 `RealtimeThread` or `NoHeapRealtimeThread`.

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

**Throws:**
- `ScopedCycleException` - If entering this ScopedMemory would violate the single parent rule.

```java
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 stack to the occurrence 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` - The memory area is not in the thread’s scope stack.

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

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

**Returns:** The `MemoryArea` of the object.

```java
public long memoryConsumed()
```
CHAPTER 5 MEMORY MANAGEMENT

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.

```java
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

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

Allocate an array of T 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:*
- `IllegalAccessException` - The class or initializer is inaccessible.
- `InstantiationException` - The array cannot be instantiated.
- `OutOfMemoryError` - Space in the memory area is exhausted.

```java
public java.lang.Object newInstance(java.lang.Class type)
```

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:*
- `IllegalAccessException` - The class or initializer is inaccessible.
- `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.
**HeapMemory**

Out Of Memory Error - Space in the memory area is exhausted.

```java
public java.lang.Object
    newInstance(java.lang.reflect.Constructor c, java.lang.Object[] args)
    throws IllegalArgumentException, 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:**
- `IllegalArgumentException` - The class or initializer is inaccessible.
- `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.

```java
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.

### 5.2 HeapMemory

**Declaration**

```java
public final class HeapMemory extends MemoryArea
```

**Description**

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

#### 5.2.1 Methods

```java
public static HeapMemory instance()
```

Returns a pointer to the singleton `HeapMemory` space.
5.3 ImmortalMemory

Declaration:

```java
public final class ImmortalMemory extends MemoryArea
```

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.

5.3.1 Methods

```java
public static ImmortalMemory instance()
```

Returns a pointer to the singleton ImmortalMemory space.

Returns: The singleton ImmortalMemory object.

5.4 SizeEstimator

Declaration:

```java
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 constructors, one can pass
SizeEstimator objects with which you can have a better feel of how big a memory area you require.

See Also: protected MemoryArea(SizeEstimator size), public LTMemory(SizeEstimator size), MemoryArea initial, SizeEstimator maximum), public VTMemory(SizeEstimator initial, SizeEstimator maximum)

5.4.1 Constructors

public SizeEstimator()

5.4.2 Methods

public long getEstimate()

Returns an estimate of the number of bytes needed to store all the objects reserved.

public void reserve(java.lang.Class c, int n)

Take into account additional n instances of Class c when estimating the size of the MemoryArea.

Parameters:
- c - The class to take into account.
- n - The number of instances of c to estimate.

public void reserve(SizeEstimator s)

Take into account an additional instance of SizeEstimator s when estimating the size of the MemoryArea.

Parameters:
- c - The class to take into account.
- n - The number of instances of c to estimate.

public void reserve(SizeEstimator s, int n)

Take into account additional n instances of SizeEstimator s when estimating the size of the MemoryArea.
CHAPTER 5  MEMORY MANAGEMENT

Parameters:
  \( c \) - The class to take into account.
  \( n \) - The number of instances of \( c \) to estimate.

5.5 ScopedMemory

**Declaration**

```java
public abstract class ScopedMemory extends MemoryArea
```

**Direct Known Subclasses:** LTMemory, VTMemory, LTPhysicalMemory, VTPhysicalMemory

**Description:**

ScopedMemory is the abstract base class of all classes dealing with representations of memory spaces with 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 `public void enter()` method for the memory area. When the last reference to the object is removed, by exiting the thread or exiting the `enter()` 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()` method of ScopedMemory is the mechanism used to activate a new memory scope. Entry into the scope is done by calling the method:

```java
public void enter(Runnable r)
```

Where \( r \) is a Runnable object whose `run()` method represents the entry point to the code that will run in the new scope. Exit from the scope occurs when the `r.run()` completes. Allocations of objects within `r.run()` are done with the ScopedMemory area. When `r.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 `public void enter()` throws ScopedCycleException.
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.
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.

5.5.1 Constructors

```java
public ScopedMemory(long size)
Create a new ScopedMemory of size size.
Parameters:
size - The size of the new ScopedMemory area in bytes. If size is less than or equal to zero an IllegalArgumentException is thrown.
```

```java
public ScopedMemory(long size, java.langRunnable r)
Create a new ScopedMemory of size size and that executes r.run() when enter() is called.
Parameters:
size - The size of the new ScopedMemory area in bytes. If size is less than or equal to zero an IllegalArgumentException is thrown.

r - The java.langRunnable whose run() method is invoked when any of the variations of enter() which do not take a java.langRunnable is called.
```

```java
public ScopedMemory(SizeEstimator size)
Create a new ScopedMemory with size equal to size.getEstimate().
```
CHAPTER 5 MEMORY MANAGEMENT

Parameters:

size - A [@link SizeEstimator] which encapsulates the size of the new ScopedMemory area.

```java
public ScopedMemory(SizeEstimator size, java.lang.Runnable r)
```

Create a new ScopedMemory with size equal to size.get() and that executes r.run() when enter() is called.

Parameters:

size - A [@link SizeEstimator] which encapsulates the size of the new ScopedMemory area.

r - The java.lang.Runnable whose run() method is invoked when any of the variations of enter() which do not take a java.lang.Runnable is called.

5.5.2 Methods

```java
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 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 or NoHeapRealtimeThread.

Overrides: public void enter() throws ScopedCycleException in class MemoryArea

Parameters:

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

Throws:

ScopedCycleException

```java
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.
java.langRunnable. 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 or NoHeapRealtimeThread.

Overrides: public void enter(java.lang.Runnable logic) throws ScopedCycleException in class MemoryArea

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

Throws:
ScopedCycleException

public long getMaximumSize()
Get the maximum size this memory area can attain. If this is a fixed size memory area, the returned value will be equal to the initial size.

Returns: The maximum size attainable.

public java.lang.Object getPortal()

Return a reference to the portal object in this instance of ScopedMemory. For a more detailed explanation of portals see public void setPortal(java.lang.Object object)

Returns: The portal object or null if there is no portal object.

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.

public void join() throws InterruptedException
Wait until the reference count of this ScopedMemory goes down to zero.

Throws:
InterruptedException - If another thread interrupts this thread while it is waiting.

public void join(HighResolutionTime time)
CHAPTER 5 MEMORY MANAGEMENT

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:

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

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 Runnable was passed, the method returns immediately.

Throws:

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

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

public void joinAndEnter(HighResolutionTime 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 java.lang.Runnable object passed at construction time. If no java.lang.Runnable was passed then this method returns immediately.
Parameters:

**time** - The time that bounds the wait.

**Throws:**

*InterruptedException* - if another thread interrupts this thread while it is waiting.

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

```java
public void joinAndEnter(java.lang.Runtime 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.Runtime object which contains the code to execute.

**time** - The time that bounds the wait.

**Throws:**

*InterruptedException* - If another thread interrupts this thread while it is waiting.

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

```java
public void joinAndEnter(java.lang.Runtime logic,
    HighResolutionTime 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.Runtime object which contains the code to execute.

**time** - The time that bounds the wait.
CHAPTER 5 MEMORY MANAGEMENT

Throws:

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

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

public void setPortal(java.lang.Object object)

Set the argument to the portal object in the memory area represented by this instance of ScopedMemory. A portal can serve as a means of interthread communication and they are used primarily when threads need to share an object that is allocated in a ScopedMemory. The portal object for a ScopedMemory must be allocated in the same ScopedMemory. Thus the following condition has to evaluate to true for the portal to be set:

this.equals(MemoryArea.getMemoryArea(object))

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.

public java.lang.String toString()

Returns a user-friendly representation of this ScopedMemory.

Overrides: java.lang.Object.toString() in class java.lang.Object

Returns: The string representation

5.6 VTMemory

Declaration

public class VTMemory extends ScopedMemory

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.
5.6.1 Constructors

```java
public VTMemory(long initialSizeInBytes,
                long maxSizeInBytes)

Creates a VTMemory of the given size.

Parameters:
  initialSizeInBytes - The size in bytes of the memory to initially allocate for this area.
  maxSizeInBytes - The maximum size in bytes this memory area can grow to.
```

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

Creates a VTMemory of the given size and logic.

Parameters:
  initialSizeInBytes - The size in bytes of the memory to initially allocate for this area.
  maxSizeInBytes - The maximum size in bytes this memory area can grow to.
  logic - The logic associated with this.
```

```java
public VTMemory(SizeEstimator initial,
                SizeEstimator maximum)

Creates a VTMemory of the given size estimated by two instances of SizeEstimator.

Parameters:
  initial - The instance of SizeEstimator which will set the initial allocation allocate for this area.
  maximum - The instance of SizeEstimator which will set the maximum allocation allocate for this area.
```

```java
public VTMemory(SizeEstimator initial,
                SizeEstimator maximum,
                java.lang.Runnable logic)
```
CHAPTER 5 MEMORY MANAGEMENT

Creates a VTMemory of the given size estimated by two instances of SizeEstimator and logic.

Parameters:
- initial - The instance of SizeEstimator which will set the initial allocation for this area.
- maximum - The instance of SizeEstimator which will set the maximum allocation for this area.
- logic - The logic associated with this.

5.6.2 Methods

public long getMaximumSize()

Return the value which defines the maximum size to which this can grow.

Overrides: public long getMaximumSize() in class ScopedMemory

public java.lang.String toString()

Overrides: public java.lang.String toString() in class ScopedMemory

5.7 LTMemory

Declaration:
public class LTMemory extends ScopedMemory

Description:
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. 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, or to enter the memory area using the public void enter() throws ScopedCycleException method within a NoHeapRealtimeThread.

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 \texttt{LTMemory} with that initial memory size. \textit{(Note: It is important to understand that the above statement does not require that if the initial memory size is $N$ and $(\text{sizeof(object1)} + \text{sizeof(object2)} + ... + \text{sizeof(objectn)} = N)$ 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. \textit{(Note: to ensure that all requested memory is available set initial and maximum to the same value)}

\textit{See Also: MemoryArea 77, ScopedMemory 84, RealtimeThread 23, NoHeapRealtimeThread 33,}

5.7.1 Constructors

\texttt{public LTMemory(long initialSizeInBytes, long maxSizeInBytes)}

Create an \texttt{LTMemory} of the given size.

\textit{Parameters:}

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

\texttt{maxSizeInBytes} - The size in bytes of the memory to allocate for this area.

\texttt{public LTMemory(long initialSizeInBytes, long maxSizeInBytes, java.lang.Runnable logic)}

Create an \texttt{LTMemory} of the given size and logic.

\textit{Parameters:}

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

\texttt{maxSizeInBytes} - The size in bytes of the memory to allocate for this area.
CHAPTER 5 MEMORY MANAGEMENT

```java
public LTMemory(SizeEstimator initial, SizeEstimator maximum)

Creates a LTMemory of the given size estimated by two instances of SizeEstimator.

Parameters:
  initial - The instance of SizeEstimator which will set the initial allocation for this area.
  maximum - The instance of SizeEstimator which will set the maximum allocation for this area.

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

Creates a LTMemory of the given size estimated by two instances of SizeEstimator and logic.

Parameters:
  initial - The instance of SizeEstimator which will set the initial allocation for this area.
  maximum - The instance of SizeEstimator which will set the maximum allocation for this area.
  logic - The logic associated with this.
```

5.7.2 Methods

```java
public long getMaximumSize()

Return the value which defines the maximum size to which this can grow.

Overrides: public long getMaximumSize() in class ScopedMemory
```

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

Overrides: public java.lang.String toString() in class ScopedMemory
```
5.8 PhysicalMemoryManager

Declaration:

```java
public final class PhysicalMemoryManager
```

Description:

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.

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`, but since filters are not part of the normative RTSJ, this exception is at best advisory.

5.8.1 Fields

```java
public static final java.lang.String ALIGNED
Specify this to identify aligned memory.

public static final java.lang.String BYTE_SWAP
Specify this if byte swapping should be used.

public static final java.lang.String DMA
Specify this to identify DMA memory.

public static final java.lang.String SHARED
Specify this to identify shared memory.
```
5.8.2 Methods

```java
public static boolean isRemovable(long address, long size)

Is the specified range of memory removable?

Parameters:
- address - 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
```

```java
public static boolean isRemoved(long address, long size)

Is any part of the specified range of memory presently removed? 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:
- address - 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
```

```java
public static void onInsertion(long base, long size, AsyncEventHandler aeh)

Register the specified AsyncEventHandler on 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 AEH.

Parameters:
- base - The starting address in physical memory
- size - The size of the memory area
- aeh - Register this AEH.

Throws:
- IllegalArgumentException - If the specified range contains no removable memory.
```

```java
public static void onRemoval(long base, long size, AsyncEventHandler 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 aeh.

Parameters:

- **base** - The starting address in physical memory
- **size** - The size of the memory area
- **aeh** - Register this aeh.

Throws:

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

```java
public static final void registerFilter(java.lang.Object name, PhysicalMemoryTypeFilter 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` - A filter for this type of memory already exists
- `RuntimeException` - The system is configured for a bounded number of filters. This filter exceeds the bound.
- `IllegalArgumentException` - The name parameter must not be an array of objects.
- `IllegalArgumentException` - The name and filter must both be in immortal memory.

```java
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.
CHAPTER 5 MEMORY MANAGEMENT

5.9 PhysicalMemoryTypeFilter

Declaration:

```java
public interface PhysicalMemoryTypeFilter {
```

5.9.1 Methods

`public boolean contains(long base, long size)`

Does the specified range of memory contain 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.

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

Search for memory of the right type.

**Parameters:**
- `base` - Start searching at this address.
- `size` - Find at least this much memory.

**Returns:** The address where memory was found or -1 if it was not found.

`public int getVMAttributes()`

Return the virtual memory attributes of this type of memory.

`public int getVMFlags()`

Return the virtual memory flags of this type of memory.

`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:

IllegalArgumentException - if the base and size do not fall into this type of memory

public boolean isPresent(long base, long size)

Checks if all of the specified range of physical memory present in the system. If any of it has been removed, false is returned.

Parameters:

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

Throws:

IllegalArgumentException - if the base and size do not fall into this type of memory

public boolean isRemovable()

If this type of memory is removable, return true.

Returns: true if this type of memory is removable.

public void onInsertion(long base, long size, AsyncEventHandler aeh)

Arrange for the specified AsyncEventHandler 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 this if any memory in the specified range is inserted.

Throws:

IllegalArgumentException - if the base and size do not fall into this type of memory

public void onRemoval(long base, long size, AsyncEventHandler aeh)

Arrange for the specified AsyncEventHandler 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 this if any memory in the specified range is removed.

Throws:
   IllegalArgumentException - if the base and size do not fall into
   this type of memory

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 - Start searching at this address.
   size - Find at least this much memory.

Returns: The address where memory was found or -1 if it was not found.

5.10 ImmortalPhysicalMemory

Declaration:
public class ImmortalPhysicalMemory extends MemoryArea

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
memory areas, and may be used in any context where ImmortalMemory is
appropriate. Objects allocated in immortal physical memory have a lifetime greater
than the application as do objects allocated in immortal memory.

5.10.1 Constructors

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

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:
   SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.
   SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory.
   UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
   MemoryTypeConflictException

public ImmortalPhysicalMemory(java.lang.Object type,
                              long base, long size)
                        throws SecurityException, SizeOutOfBoundsException,
                              OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException,
                              MemoryTypeConflictException, MemoryInUseException

Parameters:
   type - An Object representing the type of memory required (e.g., dma, shared).
   base - The physical memory address of the area.
   size - The size of the area in bytes.

Throws:
   SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
   OffsetOutOfBoundsException - The address is invalid.
   SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory.
   UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
   MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
   MemoryInUseException - The specified memory is already in use.

public ImmortalPhysicalMemory(java.lang.Object type,
                              long base, long size,
public void enter()
throws ScopedCycleException

Parameters:

- type - An Object representing the type of memory required (e.g.,
  dma, shared)
- 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

Throws:

- SecurityException - The application doesn’t have permissions to
  access physical memory or the given range of memory.
- OffsetOutOfBoundsException - The address is invalid.
- SizeOutOfBoundsException - The size is negative or extends
  into an invalid range of memory.
- UnsupportedPhysicalMemoryException - Thrown if the
  underlying hardware does not support the given type.
- MemoryTypeConflictException - The specified base does not
  point to memory that matches the request type, or if type
  specifies attributes with a conflict.
- MemoryInUseException - The specified memory is already in
  use.

public ImmortalPhysicalMemory(java.lang.Object type,
long size, java.langRunnable logic)
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException,
MemoryInUseException

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.
logic - The run() method of this object will be called whenever
public void enter()
throws ScopedCycleException
is called.

Throws:
SecurityException - The application doesn’t have permissions to
access physical memory or the given type of memory.
SizeOutOfBoundsException - The size is negative or extends
into an invalid range of memory.
UnsupportedPhysicalMemoryException - Thrown if the
underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not
point to memory that matches the request type, or if type
specifies attributes with a conflict.

public ImmortalPhysicalMemory(java.lang.Object type,
long base, SizeEstimator size)
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryType
ConflictException, MemoryInUseException

Parameters:
type - An Object representing the type of memory required (e.g.,
dma, shared)
base - The physical memory address of the area.
size - A size estimator for this memory area.

Throws:
SecurityException - The application doesn’t have permissions to
access physical memory or the given range of memory.
OffsetOutOfBoundsException - The address is invalid.
SizeOutOfBoundsException - The size is negative or extends
into an invalid range of memory.
UnsupportedPhysicalMemoryException - Thrown if the
underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not
point to memory that matches the request type, or if type
specifies attributes with a conflict.
CHAPTER 5 MEMORY MANAGEMENT

MemoryInUseException - The specified memory is already in use.

public ImmortalPhysicalMemory(java.lang.Object type, long base, SizeEstimator size, java.lang.Runnable logic)
throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:
- **type** - An Object representing the type of memory required (e.g., dma, shared)
- **base** - The physical memory address of the area.
- **size** - A size estimator for this memory area.
- **logic** - The run() method of this object will be called whenever public void enter()
  throws ScopedCycleException is called.

Throws:
- SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
- OffsetOutOfBoundsException - The address is invalid.
- SizeOutOfBoundsException - The size extends into an invalid range of memory.
- UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
- MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
- MemoryInUseException - The specified memory is already in use.

public ImmortalPhysicalMemory(java.lang.Object type, SizeEstimator size)
throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
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 - A size estimator for this area.

Throws:

SecurityException - The application doesn’t have permissions to
access physical memory or the given type of memory.

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

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

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

public ImmortalPhysicalMemory(java.lang.Object type,
SizeEstimator size, java.lang.Runnable logic)
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException

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 - A size estimator for this area.

logic - The run() method of this object will be called whenever
public void enter()
throws ScopedCycleException is called.

Throws:

SecurityException - The application doesn’t have permissions to
access physical memory or the given type of memory.

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

UnsupportedPhysicalMemoryException - Thrown if the
underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.

5.11 LTPhysicalMemory

Declaration:

```java
public class LTPhysicalMemory extends ScopedMemory
```

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`, and the same performance restrictions as `LTMemory`.

See Also: MemoryArea, ScopedMemory, VTMemory, LTMemory, VTPhysicalMemory, ImmortalPhysicalMemory, NoHeapRealtimeThread, RealtimeThread

5.11.1 Constructors

```java
public LTPhysicalMemory(java.lang.Object type, long size)
```

Throws: SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory. SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory. UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.

public LTPhysicalMemory(java.lang.Object type, long base, long size) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:

type - An Object representing the type of memory required (e.g., dma, shared)
base - The physical memory address of the area.
size - The size of the area in bytes.

Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException - The address is invalid.
SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
MemoryInUseException - The specified memory is already in use.

public LTPhysicalMemory(java.lang.Object type, long base, long size, java.lang.Runnable logic) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException
Chapter 5 Memory Management

Parameters:
   type - An Object representing the type of memory required (e.g., dma, shared)
   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:
   SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
   OffsetOutOfBoundsException - The address is invalid.
   SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory.
   UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
   MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
   MemoryInUseException - The specified memory is already in use.

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

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

Throws:
   SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.
public LTPhysicalMemory(java.lang.Object type, long base, SizeEstimator size)
  throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:
  type - An Object representing the type of memory required (e.g., dma, shared)
  base - The physical memory address of the area.
  size - A size estimator for this memory area.

Throws:
  SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
  OffsetOutOfBoundsException217 - The address is invalid.
  SizeOutOfBoundsException217 - The size extends into an invalid range of memory.
  UnsupportedPhysicalMemoryException218 - Thrown if the underlying hardware does not support the given type.
  MemoryTypeConflictException215 - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
  MemoryInUseException219 - The specified memory is already in use.

public LTPhysicalMemory(java.lang.Object type, long base, SizeEstimator size, java.lang.Runnable logic)
  throws SecurityException, SizeOutOfBoundsException
CHAPTER 5 MEMORY MANAGEMENT

import xception, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

**Parameters:**
- **type** - An Object representing the type of memory required (e.g., `dma`, `shared`)
- **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:**
- SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
- OffsetOutOfBoundsException - The address is invalid.
- SizeOutOfBoundsException - The size extends into an invalid range of memory.
- UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
- MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
- MemoryInUseException - The specified memory is already in use.

```java
public LTPhysicalMemory(java.lang.Object type,
SizeEstimator size)
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

**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** - A size estimator for this area.

**Throws:**
- SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.
SizeOutOfBoundsException - The size extends into an invalid range of memory.

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

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

public LTPhysicalMemory(java.lang.Object type, SizeEstimator size, java.langRunnable logic)
throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException

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 - A size estimator for this area.

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

 Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.

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

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

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

5.11.2 Methods

public java.lang.String toString()

Overrides: public java.lang.String toString() in class ScopedMemory
5.12 VTPhysicalMemory

Declaration:

```java
public class VTPhysicalMemory extends ScopedMemory
```

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` memory areas, and the same performance restrictions as `VTMemory`.

See Also: `MemoryArea`, `ScopedMemory`, `VTMemory`, `LTMemory`, `ImmortalPhysicalMemory`, `RealtimeThread`, `NoHeapRealtimeThread`

5.12.1 Constructors

```java
public VTPhysicalMemory(java.lang.Object type, long size)
```

Throws: `SecurityException`, `SizeOutOfBoundsException`, `UnsupportedPhysicalMemoryException`, `MemoryTypeConflictException`

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.

```java
public VTPhysicalMemory(java.lang.Object type, long base,
```

...
public VTPhysicalMemory(java.lang.Object type, long base, long size, java.langRunnable logic) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:

type - An Object representing the type of memory required (e.g., dma, shared)

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.
CHAPTER 5 MEMORY MANAGEMENT

Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.

OffsetOutOfBoundsException - The address is invalid.

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

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

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

MemoryInUseException - The specified memory is already in use.

public VTPhysicalMemory(java.lang.Object type, long size, java.lang.Runnable logic) throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException

Parameters:

type - An Object representing the type of memory required (e.g., dma, shared) is 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:

SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.

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

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

MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
public VTPhysicalMemory(java.lang.Object type, long base, SizeEstimator size) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:

type - An Object representing the type of memory required (e.g., dma, shared)
base - The physical memory address of the area.
size - A size estimator for this memory area.

Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException - The address is invalid.
SizeOutOfBoundsException - The size extends into an invalid range of memory.
UnsupportedPhysicalMemoryException - Thrown if the underlying hardware does not support the given type.
MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
MemoryInUseException - The specified memory is already in use.

public VTPhysicalMemory(java.lang.Object type, long base, SizeEstimator size, java.lang.Runnable logic) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException

Parameters:

type - An Object representing the type of memory required (e.g., dma, shared)
base - The physical memory address of the area.
size - A size estimator for this memory area.
CHAPTER 5 MEMORY MANAGEMENT

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

Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given range of memory.

OffsetOutOfBoundsException - The address is invalid.

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

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

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

MemoryInUseException - The specified memory is already in use.

public VTPhysicalMemory(java.lang.Object type, SizeEstimator size)
   throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException

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 - A size estimator for this area.

Throws:

SecurityException - The application doesn’t have permissions to access physical memory or the given type of memory.

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

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

MemoryTypeConflictException - The specified base does not point to memory that matches the request type, or if type specifies attributes with a conflict.
public VTPhysicalMemory(java.lang.Object type,
               SizeEstimator size,
               java.lang.Runnable logic)
    throws SecurityException, SizeOutOfBoundsException,
               UnsupportedPhysicalMemoryException,
               MemoryTypeConflictException

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 - A size estimator for this area.
  logic - enter this memory area with this Runnable after the
          memory area is created.

Throws:
  SecurityException - The application doesn’t have permissions to
          access physical memory or the given type of memory.
  SizeOutOfBoundsException - The size extends into an invalid
          range of memory.
  UnsupportedPhysicalMemoryException - Thrown if the
          underlying hardware does not support the given type.
  MemoryTypeConflictException - The specified base does not
          point to memory that matches the request type, or if type
          specifies attributes with a conflict.

5.12.2 Methods

public java.lang.String toString()

Overrides: public java.lang.String toString()

5.13 RawMemoryAccess

Declaration
public class RawMemoryAccess

Direct Known Subclasses: RawMemoryFloatAccess

117
Description:
An instance of RawMemoryAccess models a range of physical memory as a fixed sequence of bytes. A full complement of accessor methods allow the contents of the physical area to be accessed through offsets from the base, interpreted as byte, short, int, or long data values or as arrays of these types.

Whether the offset addresses the high-order or low-order byte is based on the value of the BYTE_ORDER static boolean variable in class RealtimeSystem.

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. 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. 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.

5.13.1 Constructors

```java
public RawMemoryAccess(java.lang.Object type, long size)
throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Creates a RawMemoryAccess object based on the parameters passed.

**Parameters:**
- `type` - An Object representing the type of memory required (e.g., dma, shared, etc) - used to define the base address and control the mapping.
- `size` - The size of the area in bytes.
Throws:

- `SecurityException` - The application doesn’t have permissions to access physical memory or the given type of memory.
- `OffsetOutOfBoundsException` - The address is invalid.
- `SizeOutOfBoundsException` - The size is negative or extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException` - Thrown if the underlying hardware does not support the given type.
- `MemoryTypeConflictException` - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

```java
public RawMemoryAccess(java.lang.Object type, long base, long size)
    throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Creates a `RawMemoryAccess` object based on the parameters passed.

Parameters:

- `type` - An Object representing the type of memory required (e.g., dma, shared, etc) - used to define the base address and control the mapping.
- `base` - The starting address for this physical memory area.
- `size` - The size of the area in bytes.

Throws:

- `SecurityException` - The application doesn’t have permissions to access physical memory or the given type of memory.
- `OffsetOutOfBoundsException` - The address is invalid.
- `SizeOutOfBoundsException` - The size is negative or extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException` - Thrown if the underlying hardware does not support the given type.
- `MemoryTypeConflictException` - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.
CHAPTER 5 MEMORY MANAGEMENT

5.13.2 Methods

public byte getByte(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get the byte at the given offset.

Parameters:
    offset - The offset at which to read the byte.

Returns: The byte read.

Throws:
    SizeOutOfBoundsException
    OffsetOutOfBoundsException

public void getBytes(long offset, byte[] bytes, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get number bytes starting at the given offset and assign them to the byte
array passed starting at position low.

Throws:
    SizeOutOfBoundsException
    OffsetOutOfBoundsException

public int getInt(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get the int at the given offset.

Parameters:
    offset - The offset at which to read the integer.

Returns: The int read.

Throws:
    SizeOutOfBoundsException
    OffsetOutOfBoundsException

public void getInts(long offset, int[] ints, int low, int number)
public long getLong(long offset)
throws OffsetOutOfBoundsException, SizeOutOfBoundsException
Get the long at the given offset.
Parameters:
offset - The offset at which to read the long.
Returns: The long read.
Throws:
SizeOutOfBoundsException
OffsetOutOfBoundsException

public void getLongs(long offset, long[] longs, int low, int number)
throws OffsetOutOfBoundsException, SizeOutOfBoundsException
Get number longs starting at the given offset and assign them to the long array passed starting at position low.
Throws:
SizeOutOfBoundsException
OffsetOutOfBoundsException

public long getMappedAddress()
Returns 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.

public short getShort(long offset)
CHAPTER 5 MEMORY MANAGEMENT

```java
public void getShorts(long offset, short[] shorts, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get number shorts starting at the given offset and assign them to the short array passed starting at position low.

Throws:
SizeOutOfBoundsException, OffsetOutOfBoundsException
```

```java
public long map()
Maps the physical memory range into virtual memory. No-op if the system doesn’t support virtual memory.
```

```java
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.
```

```java
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 - The size of the block to map in.
```
public void setByte(long offset, byte value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the byte at the given offset.

*Parameters:*
  - offset - The offset at which to write the byte.
  - value - The byte to write.

*Throws:*
  - SizeOutOfBoundsException
  - OffsetOutOfBoundsException

public void setBytes(long offset, byte[] bytes, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set number bytes starting at the given offset from the byte array passed starting at position low.

*Throws:*
  - SizeOutOfBoundsException
  - OffsetOutOfBoundsException

public void setInt(long offset, int value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the int at the given offset.

*Parameters:*
  - offset - The offset at which to write the int.
  - value - The integer to write.

*Throws:*
  - SizeOutOfBoundsException
  - OffsetOutOfBoundsException

public void setInts(long offset, int[] ints, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

123
Set number ints starting at the given offset from the int array passed starting at position low.

Throws:
    SizeOutOfBoundsException, OffsetOutOfBoundsException

public void setLong(long offset, long value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the long at the given offset.

Parameters:
    offset - The offset at which to write the long.
    value - The long to write.

Throws:
    SizeOutOfBoundsException, OffsetOutOfBoundsException

public void setLongs(long offset, long[] longs, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set number longs starting at the given offset from the long array passed starting at position low.

Throws:
    SizeOutOfBoundsException, OffsetOutOfBoundsException

public void setShort(long offset, short value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the short at the given offset.

Parameters:
    offset - The offset at which to write the short.
    value - The short to write.

Throws:
    SizeOutOfBoundsException, OffsetOutOfBoundsException
public void setShorts(long offset, short[] shorts, int low, int number) throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set number shorts starting at the given offset from the short array passed starting at position low.

Throws:
SizeOutOfBoundsException, OffsetOutOfBoundsException

public void unmap()

Unmap the physical memory range from virtual memory. No-op if the system doesn’t support virtual memory.

5.14 RawMemoryFloatAccess

Declaration:
public class RawMemoryFloatAccess extends RawMemoryAccess

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. 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. 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.

5.14.1 Constructors

public RawMemoryFloatAccess(java.lang.Object type, long size) throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
Create a `RawMemoryFloatAccess` object.

**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:**
- `SecurityException` - The application doesn’t have permissions to access physical memory or the given type of memory.
- `OutOfBoundsException` - The address is invalid.
- `SizeOutOfBoundsException` - The size is negative or extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException` - Thrown if the underlying hardware does not support the given type.
- `MemoryTypeConflictException` -

```java
public RawMemoryFloatAccess(java.lang.Object type, long base, long size)
    throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Create a `RawMemoryFloatAccess` object.

**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:**
- `SecurityException` - The application doesn’t have permissions to access physical memory or the given type of memory.
- `OutOfBoundsException` - The address is invalid.
- `SizeOutOfBoundsException` - The size is negative or extends into an invalid range of memory.
- `UnsupportedPhysicalMemoryException` - Thrown if the underlying hardware does not support the given type.
5.14.2 Methods

```java
public double getDouble(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get the double at the given offset.

Throws:
OffsetOutOfBoundsException 217 - The address is invalid.
SizeOutOfBoundsException 217 - The size is negative or extends
into an invalid range of memory.
```

```java
public void getDoubles(long offset, double[] doubles, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get number double values starting at the given offset in this, and assigns
them into the double array starting at position low.

Throws:
OffsetOutOfBoundsException 217 - The address is invalid.
SizeOutOfBoundsException 217 - The size is negative or extends
into an invalid range of memory.
```

```java
public float getFloat(long offset)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get the float at the given offset.

Throws:
OffsetOutOfBoundsException 217 - The address is invalid.
SizeOutOfBoundsException 217 - The size is negative or extends
into an invalid range of memory.
```

```java
public void getFloats(long offset, float[] floats, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException
```

CHAPTER 5 MEMORY MANAGEMENT

Get number float values starting at the given offset in this and assign them into the byte array starting at position low.

Throws:
- OffsetOutOfBoundsException
- SizeOutOfBoundsException

public void setDouble(long offset, double value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the double at the given offset.

Throws:
- OffsetOutOfBoundsException
- SizeOutOfBoundsException

public void setDoubles(long offset, double[] doubles, int low, int number)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Get number double values starting at the given offset in this, and assigns them into the double array starting at position low.

Throws:
- OffsetOutOfBoundsException
- SizeOutOfBoundsException

public void setFloat(long offset, float value)
    throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set the float at the given offset.

Throws:
- OffsetOutOfBoundsException
- SizeOutOfBoundsException
public void setFloats(long offset, float[] floats, int low, int number) throws OffsetOutOfBoundsException, SizeOutOfBoundsException

Set number float values starting at the given offset in this from the byte array starting at position low.

Throws:
OffsetOutOfBoundsException - The address is invalid.
SizeOutOfBoundsException - The size is negative or extends into an invalid range of memory.

5.15 MemoryParameters

Declaration:
public class MemoryParameters

Description:
Memory parameters can be given on the constructor of RealtimeThread and AsyncEventHandler. 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.

5.15.1 Fields

public static final long NO_MAX

Specifies no maximum limit.
5.15.2 Constructors

```java
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:
IllegalArgumentException
```

```java
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:
IllegalArgumentException
```
5.15.3 Methods

```java
public long getAllocationRate()
Get the allocation rate. Units are in bytes per second.
```

```java
public long getMaxImmortal()
Get the limit on the amount of memory the thread may allocate in the immortal area. Units are in bytes.
```

```java
public long getMaxMemoryArea()
Get the limit on the amount of memory the thread may allocate in the memory area. Units are in bytes.
```

```java
public void setAllocationRate(long allocationRate)
A 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.
```

```java
public boolean setAllocationRateIfFeasible(int allocationRate)
Change the limit on the rate of allocation in the heap. If this MemoryParameters 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.
```

```java
public boolean setMaxImmortalIfFeasible(long maximum)
```

```java
public boolean setMaxImmortalIfFeasible(int maximum)
```

```java
public boolean setMaxMemoryAreaIfFeasible(long maximum)
```
A 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:* False if any of the threads have already allocated more than the given value. In this case the call has no effect.

```java
public boolean setMaxMemoryAreaFeasible(long maximum)
```

A 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:* False if any of the threads have already allocated more than the given value. In this case the call has no effect.

### 5.16 GarbageCollector

*Declaration*

```java
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.

#### 5.16.1 Constructors

```java
public GarbageCollector()
```
5.16.2 Methods

```java
public abstract RelativeTime getPreemptionLatency()
```

Preemption latency is a measure of the maximum time a `RealtimeThread` may have to wait for the collector to reach a preemption-safe point. Instances of `RealtimeThread` are allowed to preempt the garbage collector (instances of `NoHeapRealtimeThread` preempt immediately but instances of `RealtimeThread` must wait until the collector reaches a preemption-safe point).

**Returns:** The preempting latency of this if applicable. May return 0 if there is no collector available.
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 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.

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.

### 6.1 MonitorControl

*Declaration*

```java
public abstract class MonitorControl
```

*Direct Known Subclasses:*

- `PriorityCeilingEmulation`
- `PriorityInheritance`
Description:
Abstract superclass for all monitor control policy objects.

6.1.1 Constructors

    public MonitorControl()

The default constructor.

6.1.2 Methods

    public static MonitorControl getMonitorControl()

Return the system default monitor control policy.

    public static MonitorControl getMonitorControl(java.lang.Object monitor)

Return the monitor control policy for the given object.

    public static void setMonitorControl(MonitorControl policy)

Control 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.

    public static void setMonitorControl(java.lang.Object monitor, MonitorControl monCtl)

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.

6.2 PriorityCeilingEmulation

Declaration:

```java
public class PriorityCeilingEmulation extends MonitorControl;
```

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 and PriorityInheritance.

6.2.1 Constructors

```java
public PriorityCeilingEmulation(int ceiling)
```

Create a `PriorityCeilingEmulation` object with a given ceiling.

Parameters:

- `ceiling` - Priority ceiling value.

6.2.2 Methods

```java
public int getDefaultCeiling()
```

Get the priority ceiling for this `PriorityCeilingEmulation` object.

6.3 PriorityInheritance

Declaration:

```java
public class PriorityInheritance extends MonitorControl;
```

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 and PriorityCeilingEmulation.
6.3.1 Constructors

```java
public PriorityInheritance()
```

6.3.2 Methods

```java
public static PriorityInheritance instance()
```

Return a pointer to the singleton `PriorityInheritance`.

### 6.4 WaitFreeWriteQueue

**Declaration**

```java
public class WaitFreeWriteQueue
```

**Description**

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` 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.

#### 6.4.1 Constructors

```java
public WaitFreeWriteQueue(java.lang.Thread writer,
                          java.lang.Thread reader,
                          int maximum,
                          MemoryArea memory)
```

throws `IllegalArgumentException`, `InstantiationException`, `ClassNotFoundException`, `IllegalAccessException`

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 in which this object and internal elements are allocated.

Throws: 
IllegalAccessException, ClassNotFoundException, InstantiationException, IllegalArgumentException

6.4.2 Methods

public void clear()
Set this to empty.

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 the queue was full, object was enqueued, and the last entry was overwritten with object

Throws: 
MemoryScopeException

public boolean isEmpty()
Used to determine if this is empty.

Returns: True if this is empty and false if this is not empty.

public boolean isFull()
Used to determine if this is full.

Returns: True if this is full and false if this is not full.

public java.lang.Object read()}
A synchronized read on the queue.

*Returns:* The `java.lang.Object` read or null if this is empty.

```java
public int size()
```

Used to determine the number of elements in this.

*Returns:* An integer which is the number of non-empty positions in this.

```java
public boolean write(java.lang.Object object)
```

Try to insert an element into the queue.

*Parameters:*
- `object` - The `java.lang.Object` to insert.

*Returns:* True if the insert succeeded, false if not.

*Throws:* `MemoryScopeException`

### 6.5 WaitFreeReadQueue

**Declaration**

```java
public class WaitFreeReadQueue
```

**Description**

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` 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.

### 6.5.1 Constructors

```java
public WaitFreeReadQueue(java.lang.Thread writer,
        java.lang.Thread reader, int maximum,
```
A queue with an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method. The memory areas of the given threads are found. If these memory areas are the same the queue is created in that memory area. If these memory areas are different the queue is created in the memory area accessible by the most restricted thread type.

**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` in which this object and internal elements are stored.

**Throws:**
- `IllegalAccessException`
- `ClassNotFoundException`
- `InstantiationException`
- `IllegalArgumentException`

```java
public WaitFreeReadQueue(java.lang.Thread writer, java.lang.Thread reader, int maximum, MemoryArea77 memory, boolean notify) throws IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException
```

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` in which this object and internal elements are stored.
- `notify` - Whether or not the reader is notified when data is added.

**Throws:**
- `IllegalAccessException`
- `ClassNotFoundException`
- `InstantiationException`
- `IllegalArgumentException`
6.5.2 Methods

```java
public void clear()
Set this to empty.
```

```java
public boolean isEmpty()
Used to determine if this is empty.
Returns: True if this is empty and false if this is not empty.
```

```java
public boolean isFull()
Used to determine if this is full.
Returns: True if this is full and false if this is not full.
```

```java
public java.lang.Object read()
Returns the next element in the queue unless the queue is empty. If the queue is empty null is returned.
```

```java
public int size()
Used to determine the number of elements in this.
Returns: An integer which is the number of non-empty positions in this.
```

```java
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 which has noHeap characteristics.
```

```java
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.
Throws:
MemoryScopeException
```

Obsoleted version is 1.0.1
6.6 WaitFreeDequeue

Declaration:

```java
public class WaitFreeDequeue {

Description:
The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread` and `java.lang.Thread`. See `WaitFreeWriteQueue` or `WaitFreeReadQueue` for more details. Instances of this class create a `WaitFreeWriteQueue` and a `WaitFreeReadQueue` and make calls on the respective `read()` and `write()` methods.

6.6.1 Constructors

```java
public WaitFreeDequeue(java.lang.Thread writer, java.lang.Thread reader, int maximum, MemoryArea area)
throws IllegalArgumentException, IllegalAccessException, ClassNotFoundException, InstantiationException
```

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` and the `WaitFreeWriteQueue`.
- `area` - The `MemoryArea` in which this object and internal elements are allocated.

Throws:
- `InstantiationException`, `ClassNotFoundException`, `IllegalAccessException`, `IllegalArgumentException`

6.6.2 Methods

```java
public java.lang.Object blockingRead()
```
A synchronized call of the `read()` method of the underlying `WaitFreeWriteQueue`. This call blocks on queue empty and will wait until there is an element in the queue to return.

*Returns:* An `java.lang.Object` from this.

```java
public boolean blockingWrite(java.lang.Object object)
 throws MemoryScopeException
```

A synchronized call of the `write()` method of the underlying `WaitFreeReadQueue`. 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 `object` is now in this.

*Throws:*
- `MemoryScopeException`

```java
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 `nonBlockingWrite()` call.

*Parameters:*
- `object` - The `java.lang.Object` which will overwrite the last object if this is full. Otherwise `object` will be placed in this.

```java
public java.lang.Object nonBlockingRead()
```

An unsynchronized call of the `read()` method of the underlying `WaitFreeReadQueue`. If there are no elements in this then null is returned.

```java
public boolean nonBlockingWrite(java.lang.Object object)
 throws MemoryScopeException
```

An unsynchronized call of the `write()` method of the underlying `WaitFreeWriteQueue`. 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
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.

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.

7.1 HighResolutionTime

Declaration:

```java
public abstract class HighResolutionTime implements
```
Class **HighResolutionTime** is the base class for **AbsoluteTime**, **RelativeTime**, **RationalTime**.

### 7.1.1 Methods

**public abstract AbsoluteTime absolute(Clock clock)**

Convert this time to an absolute time, relative to some clock. 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** - This clock is used to convert this time into absolute time.

**public abstract AbsoluteTime absolute(Clock clock, AbsoluteTime dest)**

Convert this time to an absolute time, relative to some clock. 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** - This clock is used to convert this time into absolute time.
- **dest** - If null, a new object is created and returned as result; else dest is returned.

**public int compareTo(HighResolutionTime time)**

Compares this HighResolutionTime with the specified HighResolutionTime.

**Parameters:**
- **time** - compares with this time.

**public int compareTo(java.lang.Object object)**
CHAPTER 7 Time

For the Comparable interface.

*Specified By:* java.lang.Comparable.compareTo(java.lang.Object) in interface java.lang.Comparable

```java
public boolean equals(HighResolutionTime time)
```

Returns true if the argument object has the same values as this.

*Parameters:*
  - `time` - Values are compared to this.

```java
public boolean equals(java.lang.Object object)
```

Returns true if the argument is a HighResolutionTime reference and has the same values as this.

*Overrides:* java.lang.Object.equals(java.lang.Object) in class java.lang.Object

*Parameters:*
  - `object` - Values are compared to this.

```java
public final long getMilliseconds()
```

Returns the milliseconds component of this.

*Returns:* The milliseconds component of the time past the epoch represented by this.

```java
public final int getNanoseconds()
```

Returns nanoseconds component of this.

```java
public int hashCode()
```

*Overrides:* java.lang.Object.hashCode() in class java.lang.Object

```java
public abstract RelativeTime relative(Clock clock)
```

Change the association of this from the currently associated clock to the given clock.

```java
public abstract RelativeTime relative(Clock clock, HighResolutionTime time)
```
Convert the given instance of HighResolutionTime to an instance of RelativeTime relative to the given instance of Clock.

```java
public void set(HighResolutionTime time)
```

Changes the time represented by the argument to some time between the invocation of the method and the return of the method.

**Parameters:**
- `time`: The HighResolutionTime which will be set to represent the current time.

```java
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 representing time before the epoch is given as a parameter to the methods.

```java
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.

```java
public static void waitForObject(java.lang.Object target, HighResolutionTime 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:**
- `InterruptedException` - If another thread interrupts this thread while it's waiting.

See Also: `java.lang.Object.wait(long)`, `java.lang.Object.wait(long)`, `java.lang.Object.wait(long, int)`

### 7.2 AbsoluteTime

**Declaration:**
```java
public class AbsoluteTime extends HighResolutionTime
```

**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 a method. For `add` and `subtract` 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.

#### 7.2.1 Constructors

```java
public AbsoluteTime()
```
AbsoluteTime

Equal to new AbsoluteTime(0,0).

```java
public AbsoluteTime(AbsoluteTime time)
```

Make a new AbsoluteTime object from the given AbsoluteTime object.

Parameters:
- `time` - The AbsoluteTime object as the source for the copy.

```java
public AbsoluteTime(java.util.Date date)
```

Equivalent to new AbsoluteTime(date.getTime(),0)

Parameters:
- `date` - The java.util.Date representation of the time past the epoch

```java
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

### 7.2.2 Methods

```java
public AbsoluteTime absolute(Clock clock)
```

Convert this time to an absolute time relative to a given clock.

Overrides: public abstract AbsoluteTime absolute(Clock clock) in class HighResolutionTime

Parameters:
- `clock` - Clock on which this is based

Returns: this

```java
public AbsoluteTime absolute(Clock clock, AbsoluteTime destination)
```

Convert this time to an absolute time. For an AbsoluteTime, this is really easy: it just return itself. Presumes that this time is already relative to the given clock.
Chapter 7 Time

Overrides: public abstract AbsoluteTime add(AbsoluteTime dest) in class HighResolutionTime

Parameters:
  clock - Clock on which this is based
  destination - Converted to an absolute time

Returns: this

public AbsoluteTime add(long millis, int nanos)  
Add millis and nanos to this. A new object is allocated for the result

Parameters:
  millis - the milliseconds value to be added to this
  nanos - the nanoseconds value to be added to this

Returns: the result after adding this with millis and nanos.

public AbsoluteTime add(long millis, int nanos, AbsoluteTime destination)
  If a destination is non-null, the result is placed there and the destination is returned. Otherwise a new object is allocated for the result.

Parameters:
  millis - milliseconds
  nanos - nanoseconds

Returns: the result

public final AbsoluteTime add(RelativeTime time)
  Return this + time. A new object is allocated for the result.

Parameters:
  time - the time to add to this

Returns: the result

public AbsoluteTime add(RelativeTime time, AbsoluteTime destination)
  Return this + time. If destination is non-null, the result is placed there and destination is returned. Otherwise a new object is allocated for the result.
Parameters:

time - the time to add to this
destination - to place the result in

Returns: the result

public java.util.Date getDate()

Returns: The time past the epoch represented by this as a java.util.Date.

public RelativeTime relative(Clock clock)

Change the association of this from the currently associated clock to the
given clock.

Overrides: public abstract RelativeTime relative(Clock clock) in class HighResolutionTime

public RelativeTime relative(Clock clock, AbsoluteTime destination)

Convert the given instance of RelativeTime to an instance of Relative-
Time relative to the given instance of Clock.

public void set(java.util.Date date)

Change the time represented by this.

Parameters:

date - java.util.Date which becomes the time represented by this
after the completion of this method.

public final RelativeTime subtract(AbsoluteTime time)

Parameters:

time - absolute time to subtract from this

Returns: this-time. A new object is allocated for the result.

public final RelativeTime subtract(AbsoluteTime time, RelativeTime destination)

Parameters:

time - absolute time to subtract from this
destination - place to store the result. New object allocated if null
CHAPTER 7  TIME

_Returns:_ this-time. A new object is allocated for the result.

```java
public final AbsoluteTime subtract(RelativeTime time)
```

_Parameters:_

time - relative time to subtract from this

_Returns:_ this-time. A new object is allocated for the result.

```java
public AbsoluteTime subtract(RelativeTime time, AbsoluteTime destination)
```

_Parameters:_

time - relative time to subtract from this

destination - place to store the result. New object allocated if null

_Returns:_ this-time. A new object is allocated for the result.

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

_Returns:_ String object converted from this.

### 7.3 RelativeTime

_Debclaration:_

```java
public class RelativeTime extends HighResolutionTime
```

_All Implemented Interfaces:_ java.lang.Comparable

_Direct Known Subclasses:_ RationalTime

_Description:_

An object that represents a time interval millis/1E3+nanos/1E9 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 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 the a method. For add and subtract 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.

7.3.1 Constructors

```java
public RelativeTime()
```
Equivalent to new RelativeTime(0,0)

```java
public RelativeTime(long millis, int nanos)
```
Construct a RelativeTime object which means a time millis milliseconds plus nanos nanoseconds past the Clock time.

**Parameters:**
- `millis` - The milliseconds component of the time past the Clock time
- `nanos` - The nanoseconds component of the time past the Clock time

```java
public RelativeTime(RelativeTime time)
```
Make a new RelativeTime object from the given RelativeTime object

**Parameters:**
- `time` - The RelativeTime object used as the source for the copy

7.3.2 Methods

```java
public AbsoluteTime absolute(Clock clock)
```
Overrides: public abstract AbsoluteTime absolute(Clock clock) in class HighResolutionTime

```java
public AbsoluteTime absolute(Clock clock, AbsoluteTime destination)
```
Convert this time to an absolute time. For a RelativeTime, this involved adding the clocks notion of now to this interval and constructing a new AbsoluteTime based on the sum

**Overrides:**
public abstract AbsoluteTime absolute(Clock clock, AbsoluteTime dest) in class HighResolutionTime
Parameters:
  clock - if null, Clock.getRealTimeClock() is used

public RelativeTime add(long millis, int nanos)

Add a specific number of milli and nano seconds to this. A new object is allocated.

Parameters:
  millis - milli seconds to add
  nanos - nano seconds to add

Returns: A new object containing the result

public RelativeTime add(long millis, int nanos, RelativeTime destination)

Add a specific number of milli and nano seconds to this. A new object is allocated if destination is null, otherwise store there.

Parameters:
  millis - milli seconds to add
  nanos - nano seconds to add
  destination - to store the result

Returns: A new object containing the result

public final RelativeTime add(RelativeTime time)

Return this + time. A new object is allocated for the result.

Parameters:
  time - the time to add to this

Returns: the result

public RelativeTime add(RelativeTime time, RelativeTime destination)

Return this + time. If destination is non-null, the result is placed there and dest is returned. Otherwise a new object is allocated for the result.

Parameters:
  time - the time to add to this
  destination - to place the result in

Returns: the result
public void addInterarrivalTo(AbsoluteTime destination)

Add this time to an AbsoluteTime. It is almost the same dest.add(this, dest) except that it accounts for (ie. divides by) the frequency. If destination is equal to null, NullPointerException is thrown.

Parameters:

public RelativeTime getInterarrivalTime()

Return the interarrival time that is the result of dividing this interval by its frequency. For a RelativeTime, and RationalTime with a frequency of 1, it just returns this. The interarrival time is necessarily an approximation.

Parameters:

destination - interarrival time is between this and the destination

Returns: interarrival time

public RelativeTime relative(Clock clock)

Change the association of this from the currently associated clock to the given clock.

Overrides: public abstract RelativeTime relative(Clock clock) in class HighResolutionTime

public RelativeTime relative(Clock clock, RelativeTime destination)

Set the time of this to the time of the given instance of RelativeTime with respect to the given instance of Clock.

public final RelativeTime subtract(RelativeTime time)

Parameters:

time - relative time to subtract from this

Returns: this-time. A new object is allocated for the result.
public RelativeTime\_156 subtract(RelativeTime\_156 time, RelativeTime\_156 destination)

*Parameters:*
  - time - relative time to subtract from this
  - destination - place to store the result. New object allocated if null

*Returns:* this-time. A new object is allocated for the result.

public java.lang.String toString()

Return a printable version of this time. Overrides: java.lang.Object.toString() in class java.lang.Object

*Overrides:* java.lang.Object.toString() in class java.lang.Object

*Returns:* String a printable version of this time.

### 7.4 RationalTime

*Declaration*

public class RationalTime extends RelativeTime\_156

*All Implemented Interfaces:* java.lang.Comparable

*Description*

An object that represents a time interval $\text{millis/1E3+nanos/1E9}$ 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 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
7.4.1 Constructors

```java
public RationalTime(int frequency)
Construct a new Object of RationalTime Equivalent to new RationalTime(1000, 0, frequency) — essentially a cycles-per-second value
```

```java
public RationalTime(int frequency, long millis, int nanos)
throws IllegalArgumentException
Construct a new Object of RationalTime. All arguments must be >= 0.
```

```java
Parameters:
frequency - The frequency value of this
millis - The milliseconds value of this
nanos - The nanoseconds value of this
```

```java
.Throws:
IllegalArgumentException
```

```java
public RationalTime(int frequency, RelativeTime interval)
throws IllegalArgumentException
Construct a new Object of RationalTime from the given RelativeTime
```

```java
Parameters:
frequency - The frequency value of this
interval - The relativeTime object used as the source for the copy
```

```java
.Throws:
IllegalArgumentException
```

7.4.2 Methods

```java
public AbsoluteTime absolute(Clock clock, AbsoluteTime destination)
Convert this time to an absolute time. For a RelativeTime, this involved adding the clocks notion of now to this interval and constructing a new AbsoluteTime based on the sum
CHAPTER 7 TIME

*Overrides:* `public AbsoluteTime absolute(Clock clock, AbsoluteTime destination) in class RelativeTime`

*Parameters:*
  - `clock` - if null, `Clock.getRealTimeClock()` is used

`public void addInterarrivalTo(AbsoluteTime destination)`

Add this time to an `AbsoluteTime`. It is almost the same as `dest.add(this, dest)` except that it accounts for (i.e., divides by) the frequency.

*Overrides:* `public void addInterarrivalTo(AbsoluteTime destination) in class RelativeTime`

*Parameters:*

`public int getFrequency()`

Return the frequency of this.

`public RelativeTime getInterarrivalTime()`

Gets the time duration between two consecutive ticks using frequency.

*Overrides:* `public RelativeTime getInterarrivalTime() in class RelativeTime`

`public RelativeTime getInterarrivalTime(RelativeTime dest)`

Gets the time duration between two consecutive ticks using frequency.

*Overrides:* `public RelativeTime getInterarrivalTime(RelativeTime destination) in class RelativeTime`

*Parameters:*
  - `dest` - Result is stored in `dest` and returned, if null new object is returned.

`public void set(long millis, int nanos) throws IllegalArgumentException`

Change the indicated interval of this to the sum of the values of the arguments.
RATIONAL.Time

Overrides: public void set(long millis, int nanos) in class HighResolutionTime

Parameters:
  millis - Millisecond part.
  nanos - Nanosecond part.

Throws:
  IllegalArgumentException

public void setFrequency(int frequency) throws ArithmeticException

Set the frequency of this.

Parameters:
  frequency - the frequency to be set for this

Throws:
  ArithmeticException
CHAPTER 7 Time

Obsolet current version is 1.0.1
Chapter 8

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.

8.1 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 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.

8.1.1 Constructors

    public Clock()

8.1.2 Methods

    public static Clock 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

    public abstract RelativeTime getResolution()
    
    Return the resolution of the clock — the interval between ticks.
    
    Returns: A RelativeTime object representing the resolution of this

    public AbsoluteTime getTime()
    
    Return the current time in a freshly allocated object.
    
    Returns: An AbsoluteTime object representing the current time.

    public abstract void getTime(AbsoluteTime time)
Return the current time in an existing object. The time represented by the given AbsoluteTime is changed some time between the invocation of the method and the return of the method.

**Parameters:**
- `time`: The AbsoluteTime object which will have its time changed. If null, then nothing happens.

```java
public abstract void setResolution(RelativeTime resolution)
```

Set the resolution of this. For some hardware clocks setting resolution impossible and if called on those nothing happens.

**Parameters:**
- `resolution`: The new resolution of this.

### 8.2 Timer

**Declaration**

```java
public abstract class Timer extends AsyncEvent
```

**Direct Known Subclasses:** `OneShotTimer`, `PeriodicTimer`

**Description**

A Timer is a timed event that measures time relative to a given Clock. This class defines basic functionality available to all timers. Applications will generally use either `PeriodicTimer` to create an event that is fired repeatedly at regular intervals, or `OneShotTimer` for an event that just fires once at a specific time. A timer is always based on a Clock, 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.

#### 8.2.1 Constructors

```java
protected Timer(HighResolutionTime t, Clock c, AsyncEventHandler handler)
```

Create a timer that fires at time `t`, according to Clock `c` and is handled by the specified handler.

**Parameters:**
- `t`: The time to fire the event. Will be converted to absolute time.
c - The clock on which to base this time. If null, the system realtime clock is used.

d - 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.

8.2.2 Methods

`public ReleaseParameters5 createReleaseParameters()`  
Create a `ReleaseParameters5` block appropriate to the timing characteristics of this event. The default is the most pessimistic: `AperiodicParameters5`. 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 that it knows the values for, like cost.

`Overrides: public ReleaseParameters5 createReleaseParameters()` in class `AsyncEvent`

`Returns: An instance of ReleaseParameters.`

`public void destroy()`  
Stop this from counting and return as many of its resources as possible back to the system.

`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 reenabled before its fire time occurs and is enabled when its fire time occurs it will fire. However, it is important to note that this method does not delay the time before a possible firing. For example, if the timer is set to fire at time 42 and the `disable()` is called at time 30 and `enable()` is called at time 40 the firing will occur at time 42 (not time 52). These semantics imply also, that firings are not queued. Using the above example, if enable was called at time 43 no firing will occur, since at time 42 this was disabled.

`public void enable()`  
Re-enable this timer after it has been disabled. See `Timer.disable()`
CHAPTER 8  TIMERS

public Clock getClock()

Return the Clock that this timer is based on

Returns: clock The clock of this timer based on

public AbsoluteTime getFireTime()

Get the time at which this event will fire

Returns: an AbsoluteTime object representing the absolute time at which
this will fire.

public boolean isRunning()

Tests this to determine if this and been started and is in a state (enabled)
such that when the given time occurs it will fire the event.

Returns: True if the timer has been started and is in the enabled state.
False, if the timer has either not been started, started and is in the
disabled state, or started and stopped.

public void reschedule(HighResolutionTime time)

Change the scheduled time for this event. can take either absolute or rela-
tive times.

Parameters:

t - the time to reschedule for this event firing if t is null, the previous
fire time is still the time at which this will fire.

public void start()

A Timer starts measuring time from when it is started

public boolean stop()

Stops a timer that is running and changes its state to not started.

Returns: True, if this was started and enabled and stops this. The new state
of this is not started. False, if this was not started or disabled.
The state of this is not changed.

8.3  OneShotTimer

Declaration

public class OneShotTimer extends Timer

170
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.

8.3.1 Constructors

public OneShotTimer(HighResolutionTime time, AsyncEventHandler handler)

Create an instance of AsyncEvent that will execute its fire method at the expiration of the given time.

Parameters:

time - After timeout time units from 'now' fire will be executed
handler - The AsyncEventHandler that will be scheduled when fire is executed

public OneShotTimer(HighResolutionTime start, Clock clock, AsyncEventHandler handler)

Create an instance of AsyncEvent, based on the given clock, that will execute 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 executed

8.4 PeriodicTimer

Declaration

public class PeriodicTimer extends Timer

Description

An AsyncEvent 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 HighResolutionTime argument types is RationalTime then the system guarantees that the fire method will be executed exactly frequency times
every unit time (see RationalTime constructors) by adjusting the interval between executions of fire(). This is similar to a thread with PeriodicParameters except that it is lighter weight. If a PeriodicTimer is disabled, it still counts, and if enabled at some later time, it will fire at its next scheduled fire time.

8.4.1 Constructors

```java
public PeriodicTimer(HighResolutionTime start, RelativeTime interval, AsyncEventHandler handler)
```

Create an instance of AsyncEvent that executes its fire method periodically

**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 executed

```java
public PeriodicTimer(HighResolutionTime start, RelativeTime interval, Clock clock, AsyncEventHandler handler)
```

Create an instance of AsyncEvent that executes its fire method periodically

**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 executed

8.4.2 Methods

```java
public ReleaseParameters createReleaseParameters()
```
Create a `ReleaseParameters` object with the next fire time as the start time and the interval of this as the period.

**Overrides:** `public ReleaseParameters createReleaseParameters()` in class `Timer`

**Returns:** an instance of `ReleaseParameters` object

```java
public void fire()
```

Causes the instance of the superclass `AsyncEvent` to occur now.

**Overrides:** `public void fire()` in class `AsyncEvent`

```java
public AbsoluteTime getFireTime()
```

Return the next time at which this will fire.

**Overrides:** `public AbsoluteTime getFireTime()` in class `Timer`

```java
public RelativeTime getInterval()
```

Return the interval of this Timer

**Returns:** a `RelativeTime` object which is the current interval of this

```java
public void setInterval(RelativeTime interval)
```

Reset the interval of this Timer

**Parameters:**
- `interval` - A `RelativeTime` object which is the interval to reset this Timer
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 \texttt{SchedulingParameters}, \texttt{ReleaseParameters}, and \texttt{MemoryParameters}. However, there is not necessarily a separate thread for each \texttt{run()} method. The class \texttt{BoundAsyncEventHandler} extends \texttt{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 \texttt{interrupt()} method in \texttt{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 \texttt{wait()}, \texttt{sleep()}, or \texttt{join()}. This specification extends the effect of \texttt{Thread.interrupt()} and adds an overloaded version in \texttt{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 \texttt{AsynchronouslyInterruptedException} (\texttt{AIE}), its subclass \texttt{Timed}, the interface \texttt{Interruptible}, and in the semantics of the interrupt methods in \texttt{Thread} and \texttt{RealtimeThread}.

A method indicates its willingness to be asynchronously interrupted by including \texttt{AIE} on its \texttt{throws} clause. If a thread is asynchronously interrupted while executing a method that identifies \texttt{AIE} on its \texttt{throws} clause, then an instance of \texttt{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 \texttt{AIE}, giving the programmer the choice of using \texttt{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.

\section*{Semantics and Requirements}

This list establishes the semantics and requirements that are applicable to \texttt{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 \texttt{AsyncEvent} occurs (by either program logic or a happening), all \texttt{run()} methods of instances of the \texttt{AsyncEventHandler} class that have been added to the instance of \texttt{AsyncEvent} by the execution of \texttt{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 \texttt{AsyncEvent} and \texttt{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. 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.

5. Constructors are allowed to include `AsynchronouslyInterruptedException` in their `throws` clause and will thus be interruptible.

6. 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:

- Control enters an AI-method.
- Control returns to an AI-method.
- 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.

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
### SEMANTICS AND REQUIREMENTS

Precedence among active instances of AIE. Let AIE\(_0\) be the AIE raised when `t.interrupt()` is invoked and AIE\(_i\) (\(i = 1,\ldots,n\), 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.
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

```java
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().
9.1 AsyncEvent

Declaration:
public class AsyncEvent

Direct Known Subclasses: Timer

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 and Scheduler).

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.

9.1.1 Constructors

public AsyncEvent()

9.1.2 Methods

public void addHandler(AsyncEventHandler handler)

Add a handler to the set of handlers associated with this event. An 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.

public void bindTo(java.lang.String happening)
throws UnknownHappeningException
Binds this to an external event (a happening). The meaningful values of \texttt{happening} are implementation dependent. This \texttt{AsyncEvent} is considered to have occurred whenever the external event occurs.

\textit{Parameters:}

\begin{itemize}
  \item \texttt{happening} - An implementation dependent value that binds this \texttt{AsyncEvent} to some external event.
\end{itemize}

\textit{Throws:}

\begin{itemize}
  \item \texttt{UnknownHappeningException} - if the happening string is not supported by the system.
\end{itemize}

\texttt{public ReleaseParameters createReleaseParameters()} \\
Create a \texttt{ReleaseParameters} block appropriate to the timing characteristics of this event. The default is the most pessimistic: \texttt{AperiodicParameters}. 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 that it knows the values for, like cost.

\texttt{public void fire()} \\
Fire (schedule the \texttt{run()} methods of) the handlers associated with this event.

\texttt{public boolean handledBy(AsyncEventHandler handler)} \\
Returns true if and only if this event is handled by this handler.

\textit{Parameters:}

\begin{itemize}
  \item \texttt{target} - The handler to be tested to determine if it is associated with this. Returns false if \texttt{target} is null.
\end{itemize}

\texttt{public void removeHandler(AsyncEventHandler handler)} \\
Remove a handler from the set associated with this event.

\textit{Parameters:}

\begin{itemize}
  \item \texttt{handler} - The handler to be disassociated from this. If null nothing happens. If not already associated with this then nothing happens.
\end{itemize}

\texttt{public void setHandler(AsyncEventHandler handler)} \\
Associate a new handler with this event, removing 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 and only handler to be associated with this. If `handler` is null then no handler will be associated with this (i.e., remove all handlers).

```java
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 AsyncEvent is bound.

**Throws:**
- `UnknownHappeningException` - if this AsyncEvent is not bound to the given `happening` or the given `happening` string is not supported by the system.

### 9.2 AsyncEventHandler

**Declaration**

```java
public class AsyncEventHandler implements Schedulable
```

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

**Direct Known Subclasses:** `BoundAsyncEventHandler`

**Description**

An asynchronous event handler encapsulates code that gets run at some time after an AsyncEvent occurs.

It is essentially a `java.lang.Runnable` with a set of parameter objects, making it very much like a `RealTimeThread`. 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.
CHAPTER 9 A SYNCHRONY

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() method.

For instances of AsyncEventHandler with a release parameter of type SporadicParameters have a list of release times which correspond to execution times of AsyncEvent.fire(). The minimum interarrival time specified in SporadicParameters is enforced as defined there. Unless the handler explicitly chooses otherwise there will be one execution of the code in handleAsyncEvent() for each entry in the list. The i th execution of handleAsyncEvent() will be released for scheduling at the time of the i th 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 their AsyncEvent 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.

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.

9.2.1 Constructors

public AsyncEventHandler()

Create a handler whose SchedulingParameters are inherited from the current thread and does not have either ReleaseParameters or MemoryParameters.

public AsyncEventHandler(boolean nonheap)

Create a handler whose parameters are inherited from the current thread, if it is a RealtimeThread, or null otherwise.

Parameters:

nonheap - A flag meaning, when true, that this will have characteristics identical to a NoHeapRealtimeThread. A false value means this will have characteristics identical to a RealtimeThread. If true and the current thread is not a NoHeapRealtimeThread or a RealtimeThread executing within a ScopedMemory or ImmortalMemory scope then an IllegalArgumentException is thrown.
public AsyncEventHandler(boolean nonheap, java.lang Runnable logic)

Create a handler whose parameters are inherited from the current thread, if it is a RealtimeThread, or null otherwise.

Parameters:
nonheap - A flag meaning, when true, that this will have characteristics identical to a NoHeapRealtimeThread. A false value means this will have characteristics identical to a RealtimeThread. If true and the current thread is not a NoHeapRealtimeThread or a RealtimeThread executing within a ScopedMemory or ImmortalMemory scope then an IllegalArgumentException is thrown.

logic - The java.lang.Runnable object whose run is executed by handleAsyncEvent.

public AsyncEventHandler(java.lang Runnable logic)

Create a handler whose SchedulingParameters are inherited from the current thread and does not have either ReleaseParameters or MemoryParameters.

Parameters:

logic - The java.lang.Runnable object whose run is executed by handleAsyncEvent.

public AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap)

Create a handler with the specified parameters.

Parameters:
scheduling - A SchedulingParameters object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters of the current thread.

release - A ReleaseParameters object which will be associated with the constructed instance of this. If null this will have no ReleaseParameters.
memory - A MemoryParameters object which will be associated with the constructed instance of this. If null this will have no MemoryParameters.

area - The MemoryArea for this AsyncEventHandler. If null, inherit the current memory area at the time of construction. The initial memory area must be a reference to a ScopedMemory or ImmortalMemory object if noheap is true.

group - A ProcessingGroupParameters object to which this will be associated. 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.

logic - The java.langRunnable object whose run is executed by handleAsyncEvent.

Throws:

{@link IllegalArgument} if the initial memory area is in heap memory, and the noheap parameter is true.

public AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap, java.langRunnable logic)

Create a handler with the specified parameters.

Parameters:

scheduling - A SchedulingParameters object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters of the current thread.

release - A ReleaseParameters object which will be associated with the constructed instance of this. If null this will have no ReleaseParameters.

memory - A MemoryParameters object which will be associated with the constructed instance of this. If null this will have no MemoryParameters.

area - The MemoryArea for this AsyncEventHandler. If null, inherit the current memory area at the time of construction. The
AsyncEventHandler

initial memory area must be a reference to a ScopedMemory or ImmortalMemory object if noheap is true.

group - A ProcessingGroupParameters object to which this will be associated. 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.

Throws:
{ @link IllegalArgumentException} if the initial memory area is in heap memory, and the noheap parameter is true.

Create a handler with the specified parameters.

Parameters:

release - A ReleaseParameters object which will be associated with the constructed instance of this. If null this will have no ReleaseParameters.

scheduling - A SchedulingParameters object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters of the current thread.

memory - A MemoryParameters object which will be associated with the constructed instance of this. If null this will have no MemoryParameters.

area - The MemoryArea for this. If null the memory area will be that of the current thread.

group - A ProcessingGroupParameters object to which this will be associated. If null this will not be associated with any processing group.

logic - The java.lang.Runnable object whose run is executed by handleAsyncEvent.
9.2.2 Methods

**public boolean addIfFeasible()**

Add to the feasibility of the associated scheduler if the resulting feasibility is schedulable. If successful return true, if not return false. If there is not assigned scheduler false is returned.

**public boolean addToFeasibility()**

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of SchedulingParameters, ReleaseParameters, MemoryParameters, and ProcessingGroupParameters) of this instance of Schedulable will be considered in the feasibility analysis of the associated Scheduler until further notice. Whether the resulting system is feasible or not, the addition is completed.

*Specified By:* public boolean addToFeasibility() in interface Schedulable

*Returns:* true If the resulting system is feasible.

**protected final int getAndClearPendingFireCount()**

Atomically set to zero the number of pending executions of this handler and returns the value from before it was cleared. This is used in handlers that can handle multiple firings and that want to collapse them together. The general form for using this is:

```java
public void handleAsyncEvent() {
    int fireCount = getAndClearPendingFireCount();
    <handle the events>
}
```

*Returns:* The pending fire count.

**protected int getAndDecrementPendingFireCount()**

Atomically decrements the number of pending executions of this handler (if it was non-zero) and returns the value from before the decrement. This can be used in the handleAsyncEvent() method in this form 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 pending fire count.

    protected int getAndIncrementPendingFireCount() {

        Atomically increments the number of pending executions of this handler
        and returns the value from before the increment. The handleAsyncEvent() method does not need to do this, since the surrounding framework guarantees that the handler will be re-executed the appropriate number of times. It is only of value when there is common setup code that is expensive.

        Returns: The pending fire count.

    public MemoryArea getMemoryArea() {

        Get the current memory area.

        Returns: The current memory area in which allocations occur.

    public MemoryParameters getMemoryParameters() {

        Get the memory parameters associated with this handler.

        Returns: The MemoryParameters object associated with this.

    protected final int getPendingFireCount() {

        Return the number of pending executions of this handler

        Returns: The pending fire count.

    public ProcessingGroupParameters getProcessingGroupParameters() {

        Returns a reference to the ProcessingGroupParameters object.
CHAPTER 9 ASYNCHRONY

Specified By: public ProcessingGroupParameters getProcessingGroupParameters() in interface Schedulable

public ReleaseParameters getReleaseParameters()

Get the release parameters associated with this handler.

Specified By: public ReleaseParameters getReleaseParameters() in interface Schedulable

Returns: The ReleaseParameters object associated with this.

public Scheduler getScheduler()

Return the Scheduler for this handler.

Specified By: public Scheduler getScheduler() in interface Schedulable

Returns: The instance of the scheduler managing this.

public SchedulingParameters getSchedulingParameters()

Returns a reference to the scheduling parameters object.

Specified By: public SchedulingParameters getSchedulingParameters() in interface Schedulable

Returns: The SchedulingParameters object associated with this.

public void handleAsyncEvent()

If this handler was constructed using a separate Runnable logic object, then that Runnable object's run method is called; This method will be invoked repeatedly while fireCount is greater than zero.

public boolean removeFromFeasibility()

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

Specified By: public boolean removeFromFeasibility() in interface Schedulable

Returns: true If the resulting system is feasible.

public final void run()

Used by the asynchronous event mechanism, see AsyncEvent. 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: java.lang.Runnable.run() in interface java.langRunnable

public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

public boolean setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

public boolean setIfFeasible(ReleaseParameters release, ProcessingGroupParameters group)

Returns true if, after considering the values of the parameters, the task set would still be feasible. In this case the values of the parameters are changed. Returns false if, after considering the values of the parameters, the task set would not be feasible. In this case the values of the parameters are not changed.

public void setMemoryParameters(MemoryParameters memory)
Set the memory parameters associated with this handler. When it is next fired, the executing thread will use these parameters to control memory allocation. Does not affect the current invocation of the run() of this handler.

Specified By: public void

```
setMemoryParameters(MemoryParameters memory)
```

in interface Schedulable

Parameters:

- memory - A MemoryParameters object which will become the MemoryParameters associated with this after the method call.

```java
public boolean setMemoryParametersIfFeasible(MemoryParameters memory)
```

Specified By: public boolean

```java
setMemoryParametersIfFeasible(MemoryParameters memParam)
```

in interface Schedulable

```
public void setProcessingGroupParameters(ProcessingGroupParameters group)
```

Sets the reference to the ProcessingGroupParameters object.

Specified By: public void

```java
setProcessingGroupParameters(ProcessingGroupParameters groupParameters)
```

in interface Schedulable

```
public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters group)
```

Specified By: public boolean

```java
setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)
```

in interface Schedulable

```
public void setReleaseParameters(ReleaseParameters release)
```

192
Set the release parameters associated with this handler. When it is next fired, the executing thread will use these parameters to control scheduling. If the scheduling parameters of a handler is set to null, the handler will be executed immediately when it is fired, in the thread of the firer. 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.

```java
public void setReleaseParameters(ReleaseParameters release)
```

**Parameters:**
- release - A `ReleaseParameters` object which will become the `ReleaseParameters` associated with this after the method call.

```java
public boolean setReleaseParametersIfFeasible(ReleaseParameters release)
```

**Specified By:**
- `setReleaseParametersIfFeasible(ReleaseParameters release)` in interface `Schedulable`

```java
public void setScheduler(Scheduler scheduler)
```

**Specified By:**
- `setScheduler(Scheduler scheduler)` in interface `Schedulable`

**Parameters:**
- scheduler - An instance of `Scheduler` (or subclasses) which will manage the execution of this thread. If `scheduler` is null nothing happens.

**Throws:**
- `IllegalThreadStateException`

```java
public void setScheduler(Scheduler scheduler,
```

**Throws:**
- `IllegalThreadStateException`
Set the scheduler for this handler. A reference to the scheduler which will manage the execution of this thread.

**Specified By:**

```java
public void setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup) throws IllegalThreadStateException
```

**Parameters:**

- `scheduler` - An instance of `Scheduler` (or subclasses) which will manage the execution of this thread. If `scheduler` is null nothing happens.
- `scheduling` - A `SchedulingParameters` object which will be associated with the constructed instance of this. If null this will be assigned the reference to the `SchedulingParameters` of the current thread.
- `release` - A `ReleaseParameters` object which will be associated with the constructed instance of this. If null this will have no `ReleaseParameters`.
- `memory` - A `MemoryParameters` object which will be associated with the constructed instance of this. If null this will have no `MemoryParameters`.
- `group` - A `ProcessingGroupParameters` object to which this will be associated. If null this will not be associated with any processing group.

**Throws:**

- `IllegalThreadStateException`

```java
public void setSchedulingParameters(SchedulingParameters scheduling)
```
BOUNDASYNCEventHandler

Set the scheduling parameters associated with this handler. When it is next fired, the executing thread will use these parameters to control scheduling. Does not affect the current invocation of the run() of this handler.

**Specified By:** public void

```java
setSchedulingParameters(SchedulingParameters scheduling) in interface Schedulable
```

**Parameters:**

- **parameters** - A SchedulingParameters object which will become the SchedulingParameters object associated with this after the method call.

```java
public boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)
```

Set the SchedulingParameters of this scheduable object only if the resulting task set is feasible.

**Specified By:** public boolean

```java
setSchedulingParametersIfFeasible(SchedulingParameters scheduling) in interface Schedulable
```

**Parameters:**

- **scheduling** - The SchedulingParameters object. If null nothing happens.

### 9.3 BoundAsyncEventHandler

**Declaration**

```java
public abstract class BoundAsyncEventHandler extends AsyncEventHandler
```

**All Implemented Interfaces:** java.langRunnable, Schedulable

**Description**

A bound asynchronous event handler is an asynchronous event handler 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.
9.3.1 Constructors

public BoundAsyncEventHandler()

Create a handler whose parameters are inherited from the current thread, if it is a RealtimeThread, or null otherwise.

public BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap, java.lang.Runnable logic)

Create a handler with the specified ReleaseParameters and MemoryParameters.

Parameters:

- scheduling - A SchedulingParameters object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters of the current thread.
- release - The ReleaseParameters object for this. A value of null will construct this without a ReleaseParameters object.
- memory - The MemoryParameters object for this. A value of null will construct this without a MemoryParameters object.
- area - The MemoryArea for this BoundAsyncEventHandler. If null, inherit the current memory area at the time of construction. The initial memory area must be a reference to a ScopedMemory or ImmortalMemory object if nonheap is true.
- nonheap - A flag meaning, when true, that this will have characteristics identical to a NoHeapRealtimeThread.
- group - A ProcessingGroupParameters object to which this will be associated. If null this will not be associated with any processing group.
- logic - The java.lang.Runnable object whose run is executed by handleAsyncEvent.
9.4 Interruptible

**Declaration**

```
public interface Interruptible
```

**Description**

Interruptible is an interface implemented by classes that will be used as arguments on the `doInterruptible()` of `AsynchronouslyInterruptedException` and its subclasses. `doInterruptible()` invokes the implementation of the method in this interface. Thus the system can ensure correctness before invoking run() and correctly cleaned up after run() returns.

### 9.4.1 Methods

**public void interruptAction(AsynchronouslyInterruptedException 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` from within the `interruptAction()` method.

**public void run(AsynchronouslyInterruptedException exception)**

```
throws AsynchronouslyInterruptedException
```

The main piece of code that is executed when an implementation 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 from within
the run() method.

Throws:

AsynchronouslyInterruptedException

9.5 AsynchronouslyInterruptedException

Declaration:

public class AsynchronouslyInterruptedException extends java.lang.InterruptedException

All Implemented Interfaces: java.io.Serializable

Direct Known Subclasses: Timed

Description:

An special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a RealtimeThread.

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(), public void interrupt(), 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
A SYNCHRONOUSLY INTERRUPTED EXCEPTION

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) will generate a specific instance of
AsynchronouslyInterruptedException which applications can identify and thus
limit propagation.

9.5.1 Constructors

public AsynchronouslyInterruptedException()

Create an instance of AsynchronouslyInterruptedException.

9.5.2 Methods

public boolean disable()

Defer the throwing of this exception. If interrupt() 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 otherwise returns false.

public boolean doInterruptible(Interruptible logic)

Execute the run() method of the given Interruptible. This method
may be on the stack in exactly one RealtimeThread. 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:

code - An instance of an Interruptible whose run() method
will be called.
CHAPTER 9  ASYNCHRONY

Returns:  True if the method call completed normally. Returns false if another call to doInterruptible has not completed.

public boolean enable()

Enable the throwing of this exception. This is valid only within a call to doInterruptible(). Otherwise it returns false and does nothing.

Returns:  True if this is enabled otherwise returns false.

public boolean fire()

Make this exception the current exception if doInterruptible has been invoked and not completed.

Returns:  True if this was fired. If there is no current invocation of doInterruptible, then false is returned with no other effect. False is also returned if there is already a current doInterruptible or if disable has been called.

public static AsynchronouslyInterruptedException getGeneric()

Return the system generic AsynchronouslyInterruptedException, which is generated when RealtimeThread.interrupt is invoked.

public boolean happened(boolean propagate)

Used with an instance of this exception to see if the current exception is this exception.

Parameters:
propagate - Propagate the exception if true and this exception is not the current one. 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. Returns false if this is not the current exception.

public boolean isEnabled()

Query the enabled status of this exception.

Returns:  True if this is enabled otherwise returns false.

public static void propagate()
Cause the pending exception to continue up the stack.

9.6 Timed

Declaration:
public class Timed extends AsynchronouslyInterruptedException

All Implemented Interfaces: java.io.Serializable

Description:
Create a scope in a RealtimeThread 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);

9.6.1 Constructors

public Timed(HighResolutionTime time)
throws IllegalArgumentException

Create an instance of Timed with a timer set to timeout. If the time is in the past the AsynchronouslyInterruptedException 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:
IllegalArgumentException

9.6.2 Methods

public boolean doInterruptible(Interruptible logic)
Execute a timeout method. Starts the timer and executes the `run()` method of the given `Interruptible` object.

*Overrides:* `public boolean doInterruptible(Interruptible logic)` in class `AsynchronouslyInterruptedException`

**Parameters:**
- `logic` - Implements an `Interruptible` `run()` method. If null nothing happens.

`public void resetTime(HighResolutionTime 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.
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.
**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.

### 10.1 POSIXSignalHandler

**Declaration**

```java
public final class POSIXSignalHandler
```

**Description**

Use instances of AsyncEvent to handle POSIX signals. Usage:

```java
POSIXSignalHandler.addHandler(SIGINT, intHandler);
```

This class is required to be implemented only if the underlying operating system supports POSIX signals.

#### 10.1.1 Fields

```java
public static final int SIGABRT
```

Used by abort, replace SIGIOT in the future.

```java
public static final int SIGALRM
```

Alarm clock.

```java
public static final int SIGBUS
```

Bus error.

```java
public static final int SIGCANCEL
```

Thread cancellation signal used by libthread.

```java
public static final int SIGCHLD
```
Child status change alias (POSIX).

```java
public static final int SIGCLD
Child status change.
```

```java
public static final int SIGCONT
Stopped process has been continued.
```

```java
public static final int SIGEMT
EMT instruction.
```

```java
public static final int SIGFPE
Floating point exception.
```

```java
public static final int SIGFREEZE
Special signal used by CPR.
```

```java
public static final int SIGHUP
Hangup.
```

```java
public static final int SIGILL
Illegal instruction (not reset when caught).
```

```java
public static final int SIGINT
Interrupt (rubout).
```

```java
public static final int SIGIO
Socket I/O possible (SIGPOLL alias).
```

```java
public static final int SIGIOT
IOT instruction.
```

```java
public static final int SIGKILL
Kill (cannot be caught or ignored).
```
CHAPTER 10 SYSTEM AND OPTIONS

```java
public static final int SIGLOST
    Resource lost (e.g., record-lock lost).

public static final int SIGLP
    Special signal used by thread library.

public static final int SIGPIPE
    Write on a pipe with no one to read it.

public static final int SIGPOLL
    Pollable event occurred.

public static final int SIGPROF
    Profiling timer expired.

public static final int SIGPWR
    Power-fail restart.

public static final int SIGQUIT
    Quit (ASCII FS).

public static final int SIGSEGV
    Segmentation violation.

public static final int SIGSTOP
    Stop (cannot be caught or ignored).

public static final int SIGSYS
    Bad argument to system call.

public static final int SIGTERM
    Software termination signal from kill.
```

```java
public static final int SIGTHAW
```
Special signal used by CPR.

public static final int SIGTRAP
Trace trap (not reset when caught).

public static final int SIGTSTP
User stop requested from tty.

public static final int SIGTIN
Background tty read attempted.

public static final int SIGTOU
Background tty write attempted.

public static final int SIGURG
Urgent socket condition.

public static final int SIGUSR1
User defined signal = 1.

public static final int SIGUSR2
User defined signal = 2.

public static final int SIGVTALRM
Virtual timer expired.

public static final int SIGWAVING
Process’s lwps are blocked.

public static final int SIGWINCH
Window size change.

public static final int SIGXCPU
Exceeded cpu limit.
CHAPTER 10 SYSTEM AND OPTIONS

public static final int SIGXFSZ

Exceeded file size limit.

10.1.2 Constructors

public POSIXSignalHandler()

10.1.3 Methods

public static void addHandler(int signal, AsyncEventHandler handler)

Add the given AsyncEventHandler to the list of handlers of the AsyncEvent 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 IllegalArgumentException will be thrown.

handler - An AsyncEventHandler which will be scheduled when the given signal occurs.

public static void removeHandler(int signal, AsyncEventHandler handler)

Remove the given AsyncEventHandler to the list of handlers of the AsyncEvent 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 IllegalArgumentException will be thrown.

handler - An AsyncEventHandler which will be scheduled when the given signal occurs.

public static void setHandler(int signal, AsyncEventHandler handler)

Set the given AsyncEventHandler as the handler of the AsyncEvent of the given signal.
10.2 RealtimeSecurity

Declaration:

```java
public class RealtimeSecurity
```

Description:
Security policy object for real-time specific issues. Primarily used to control access to physical memory.

10.2.1 Constructors

```java
public RealtimeSecurity()
```

10.2.2 Methods

```java
public void checkAccessPhysical()
```

Throws: `SecurityException` - the application doesn’t have permission.

```java
public void checkAccessPhysicalRange(long base, long size)
```

Throws: `SecurityException` - the application doesn’t have permission.

```java
public void checkSetFilter()
```
10.3 RealtimeSystem

Declaration:

```java
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.

10.3.1 Fields

```java
public static final byte BIG_ENDIAN
public static final byte BYTE_ORDER
public static final byte LITTLE_ENDIAN
```

10.3.2 Constructors

```java
public RealtimeSystem()
```
10.3.3 Methods

public static GarbageCollector currentGC()

Return a reference to the currently active garbage collector for the heap.

Returns: A GarbageCollector object which is the current collector collecting objects on the traditional Java heap.

public static int getConcurrentLocksUsed()

Get 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 int whose value is the number of locks in use at the time of the invocation of the method.

public static int getMaximumConcurrentLocks()

Get the maximum number of locks that can be used concurrently without incurring an execution time increase as set by the setMaximumConcurrentLocks() methods.

Returns: An int whose value is the maximum number of locks that can be in simultaneous use.

public static RealtimeSecurity getSecurityManager()

Get a reference to the security manager used to control access to real-time system features such as access to physical memory.

Returns: A RealtimeSecurity object representing the default real-time security manager.

public static void setMaximumConcurrentLocks(int numLocks)

Set 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.
public static void setMaximumConcurrentLocks(int number, boolean hard)

Set 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 is thrown.

public static void setSecurityManager(RealtimeSecurityManager manager)

Set a new real-time security manager.

Parameters:
- manager - A RealtimeSecurityManager object which will become the new security manager.

Throws:
- SecurityException - Thrown if security manager has already been set.
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 `AsynchronouslyInterruptedException` is subject to receiving an instance of `AsynchronouslyInterruptedException`.
CHAPTER 11 EXCEPTIONS

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.

11.1 DuplicateFilterException

Declaration:

```java
public class DuplicateFilterException extends java.lang.Exception
```

All Implemented Interfaces: java.io.Serializable

Description:

PhysicalMemoryManager

95 can only accommodate one filter object for each type of memory. It throws this exception if an attempt is made to register more than one filter for a type of memory.

11.1.1 Constructors

```java
public DuplicateFilterException()
```

```java
public DuplicateFilterException(java.lang.String s)
```

Parameters:
s - Detail string

11.2 InaccessibleAreaException

Declaration:

```java
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.

11.2.1 Constructors

```java
public InaccessibleAreaException()
    A constructor for InaccessibleAreaException.

public InaccessibleAreaException(java.lang.String description)
    A descriptive constructor for InaccessibleAreaException.
Parameters:
    description - Description of the error.
```

11.3 MemoryTypeConflictException

Declaration:
```
public class MemoryTypeConflictException extends java.lang.Exception
```

All Implemented Interfaces: java.io.Serializable

Description:
This exception is thrown when the PhysicalMemoryManager 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.

11.3.1 Constructors

```
public MemoryTypeConflictException()

public MemoryTypeConflictException(java.lang.String s)
Parameters:
    s - Detail string
```
11.4 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.

11.4.1 Constructors

```
public MemoryScopeException()
    A constructor for MemoryScopeException.

public MemoryScopeException(java.lang.String description)
    A descriptive constructor for MemoryScopeException.
    Parameters:
    description - A description of the exception.
```

11.5 MITViolationException

Declaration

```
public class MITViolationException extends java.lang.Exception
```

All Implemented Interfaces: java.io.Serializable

Description

Thrown by the `fire()` method of an instance of `AsyncEvent` when the bound instance of `AsyncEventHandler` with `ReleaseParameters` of type `SporadicParameters` has `mitViolationException` behavior and the minimum interarrival time gets violated.

11.5.1 Constructors

```
public MITViolationException()
```
A constructor for MITViolationException.

```java
public MITViolationException(java.lang.String description)
```

A descriptive constructor for MITViolationException.

Parameters:
- `description` - Description of the error.

### 11.6 OffsetOutOfBoundsException

**Declaration**:  
```java
public class OffsetOutOfBoundsException extends java.lang.Exception
```

**All Implemented Interfaces**: java.io.Serializable

**Description**: Thrown if the constructor of an ImmortalPhysicalMemory, LTPhysicalMemory, VTPhysicalMemory, RawMemoryAccess, or RawMemoryFloatAccess is given an invalid address.

### 11.6.1 Constructors

```java
public OffsetOutOfBoundsException()
```

```java
public OffsetOutOfBoundsException(java.lang.String description)
```

### 11.7 SizeOutOfBoundsException

**Declaration**:  
```java
public class SizeOutOfBoundsException extends java.lang.Exception
```

**All Implemented Interfaces**: java.io.Serializable

**Description**: Thrown if the constructor of an ImmortalPhysicalMemory, LTPhysicalMemory, VTPhysicalMemory, 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.

11.7.1 Constructors

public SizeOutOfBoundsException()
A constructor for SizeOutOfBoundsException.

public SizeOutOfBoundsException(java.lang.String description)
A descriptive constructor for SizeOutOfBoundsException.
Parameters:
description - The description of the exception.

11.8 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, RawMemoryFloatAccess, ImmortalPhysicalMemory, LTPhysicalMemory, VTPhysicalMemory

11.8.1 Constructors

public UnsupportedPhysicalMemoryException()
A constructor for UnsupportedPhysicalMemoryException.

public UnsupportedPhysicalMemoryException(java.lang.String description)
A descriptive constructor for `UnsupportedPhysicalMemoryException`.

**Parameters:**
- `description` - The description of the exception.

## 11.9 MemoryInUseException

**Declaration**

```java
public class MemoryInUseException extends RuntimeException
```

**All Implemented Interfaces:** java.io.Serializable

**Description:**
Thrown when an attempt is made to allocate a range of physical or virtual memory that is already in use.

### 11.9.1 Constructors

```java
public MemoryInUseException()
public MemoryInUseException(java.lang.String s)
```

**Parameters:**
- `s` - Detail string

## 11.10 ScopedCycleException

**Declaration**

```java
public class ScopedCycleException extends RuntimeException
```

**All Implemented Interfaces:** java.io.Serializable

**Description:**
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).
11.10.1 Constructors

```java
public class ScopedCycleException {
    public ScopedCycleException()
    public ScopedCycleException(String description)
}
```

11.11 UnknownHappeningException

Declaration:
```java
public class UnknownHappeningException extends RuntimeException
```

All Implemented Interfaces: java.io.Serializable

Description:
Thrown when `bindTo()` is called with an illegal happening.

11.11.1 Constructors

```java
public UnknownHappeningException()
public UnknownHappeningException(String description)
```

11.12 IllegalAssignmentError

Declaration:
```java
public class IllegalAssignmentError extends 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 `ScopedMemory` to a field in an object in `ImmortalMemory`.

11.12.1 Constructors

```java
public IllegalAssignmentError()
```
A constructor for `IllegalAssignmentError`.

```java
public IllegalAssignmentError(java.lang.String description)
```

A descriptive constructor for `IllegalAssignmentError`.

**Parameters:**

- `description` - Description of the error.

### 11.13 MemoryAccessError

**Declaration:**

```java
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`. For example, this will be thrown if logic in a `NoHeapRealtimeThread` attempts to refer to an object in the traditional Java heap.

#### 11.13.1 Constructors

```java
public MemoryAccessError()
```

A constructor for `MemoryAccessError`.

```java
public MemoryAccessError(java.lang.String description)
```

A descriptive constructor for `MemoryAccessError`.

**Parameters:**

- `description` - Description of the error.

### 11.14 ResourceLimitError

**Declaration:**

```java
public class ResourceLimitError extends java.lang.Error
```

**All Implemented Interfaces:** `java.io.Serializable`
CHAPTER 11 EXCEPTIONS

Description:
Thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks.

11.14.1 Constructors

public ResourceLimitError()
A constructor for ResourceLimitError.

public ResourceLimitError(java.lang.String description)
A descriptive constructor for ResourceLimitError.
Parameters:
description - The description of the exception.

11.15 ThrowBoundaryError

Declaration:
public class ThrowBoundaryError extends java.lang.Error
All Implemented Interfaces: java.io.Serializable

Description:
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).

11.15.1 Constructors

public ThrowBoundaryError()
A constructor for ThrowBoundaryError.

public ThrowBoundaryError(java.lang.String description)
A descriptive constructor for ThrowBoundaryError.
Parameters:
description - Description of the error.
Obsolete current version is 1.0.1
### ALMANAC LEGEND

The almanac presents classes and interfaces in alphabetic order, regardless of their package. Fields, methods and constructors are in alphabetic order in a single list.

This almanac is modeled after the style introduced by Patrick Chan in his excellent book *Java Developers Almanac*.

1. Name of the class, interface, nested class or nested interface. Interfaces are italic.
2. Name of the package containing the class or interface.
3. Inheritance hierarchy. In this example, `RealtimeThread` extends `Thread`, which extends `Object`.
4. Implemented interfaces. The interface is to the right of, and on the same line as, the class that implements it. In this example, `Thread` implements `Runnable`, and `RealtimeThread` implements `Schedulable`.
5. The first column above is for the value of the `@since` comment, which indicates the version in which the item was introduced.
6. The second column above is for the following icons. If the “protected” symbol 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.

<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Access Modifiers</th>
<th>Constructors and Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>◯ abstract</td>
<td>◆ protected</td>
<td>* constructor</td>
</tr>
<tr>
<td>● final</td>
<td></td>
<td>* field</td>
</tr>
<tr>
<td>□ static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>▴ static final</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 12
### Almanac

<table>
<thead>
<tr>
<th>AbsoluteTime</th>
<th>Comparable</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>HighResolutionTime</td>
<td>AbsoluteTime</td>
</tr>
<tr>
<td></td>
<td>AbsoluteTime</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AbsoluteTime</th>
<th>absolute(Clock clock)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteTime</td>
<td>absolute(Clock clock, AbsoluteTime destination)</td>
</tr>
<tr>
<td></td>
<td>AbsoluteTime()</td>
</tr>
<tr>
<td></td>
<td>AbsoluteTime(AbsoluteTime time)</td>
</tr>
<tr>
<td></td>
<td>AbsoluteTime(java.util.Date date)</td>
</tr>
<tr>
<td></td>
<td>AbsoluteTime(long millis, int nanos)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AbsoluteTime</th>
<th>add(long millis, int nanos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteTime</td>
<td>add(long millis, int nanos, AbsoluteTime destination)</td>
</tr>
<tr>
<td>AbsoluteTime</td>
<td>add(RelativeTime time)</td>
</tr>
<tr>
<td>AbsoluteTime</td>
<td>add(RelativeTime time, AbsoluteTime destination)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.util.Date</th>
<th>getDate()</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelativeTime</td>
<td>relative(Clock clock)</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>relative(Clock clock, AbsoluteTime destination)</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>set(java.util.Date date)</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>subtract(AbsoluteTime time)</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>subtract(AbsoluteTime time, RelativeTime destination)</td>
</tr>
</tbody>
</table>
AbsoluteTime subtract(RelativeTime time)

AbsoluteTime subtract(RelativeTime time, AbsoluteTime destination)

String toString()

AperiodicParameters

Object ➤ ReleaseParameters ➤ AperiodicParameters

AperiodicParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)

boolean setIfFeasible(RelativeTime cost, RelativeTime deadline)

AsyncEvent

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

Object ➤ AsyncEventHandler ➤ Schedulable

boolean addIfFeasible()

boolean addToFeasibility()

AsyncEventHandler()

AsyncEventHandler(boolean nonheap)

AsyncEventHandler(boolean nonheap, Runnable logic)

AsyncEventHandler(Runnable logic)
AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, 
MemoryParameters memory, MemoryArea area, 
ProcessingGroupParameters group, 
boolean nonheap)

AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, 
MemoryParameters memory, MemoryArea area, 
ProcessingGroupParameters group, 
boolean nonheap, Runnable logic)

AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, 
MemoryParameters memory, MemoryArea area, 
ProcessingGroupParameters group, 
Runnable logic)

int getAndClearPendingFireCount()

int getAndDecrementPendingFireCount()

int getAndIncrementPendingFireCount()

MemoryArea getMemoryArea()

MemoryParameters getMemoryParameters()

int getPendingFireCount()

ProcessingGroupParameters getProcessingGroupParameters()

ReleaseParameters getReleaseParameters()

Scheduler getScheduler()

SchedulingParameters getSchedulingParameters()

void handleAsyncEvent()

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(Processing GroupParameters group)

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 processingGroup)
    throws IllegalThreadStateException

void setSchedulingParameters(SchedulingParameters scheduling)

boolean setSchedulingParametersIfFeasible(SchedulingParameters sched)

AsynchronouslyInterruptedException

Object
    Throwable
        java.io.Serializable
            Exception
                InterruptedException
                    AsynchronouslyInterruptedException

Leod
    AsynchronouslyInterruptedException()

    boolean disable()
    boolean doInterruptible(Interruptible logic)
    boolean enable()
    boolean fire()

    AsynchronouslyInterruptedException getGeneric()

    boolean happened(boolean propagate)
    boolean isEnabled()
    void propagate()

BoundAsyncEventHandler

Object
    AsyncEventHandler
        Schedulable
            BoundAsyncEventHandler

Leod
    BoundAsyncEventHandler()

    BoundAsyncEventHandler(SchedulingParameters scheduling,
        ReleaseParameters release,
        MemoryParameters memory, MemoryArea area,
        ProcessingGroupParameters group,
        boolean nonheap, Runnable logic)
Clock

Object

- Clock
  - Clock()
  - getRealtimeClock()
  - getResolution()
  - getTime()
  - void getTime(AbsoluteTime time)
  - void setResolution(RelativeTime resolution)

DuplicateFilterException

Object

- Throwable
  - Exception
    - java.io.Serializable
- DuplicateFilterException
  - DuplicateFilterException()
  - DuplicateFilterException(String s)

GarbageCollector

Object

- GarbageCollector
  - GarbageCollector()
  - RelativeTime getPreemptionLatency()

HeapMemory

Object

- MemoryArea
  - HeapMemory
  - instance()
    - long memoryConsumed()
    - long memoryRemaining()
## HighResolutionTime

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsoluteTime</td>
<td>absolute(Clock clock)</td>
</tr>
<tr>
<td>AbsoluteTime</td>
<td>absolute(Clock clock, AbsoluteTime dest)</td>
</tr>
<tr>
<td>int</td>
<td>compareTo(HighResolutionTime time)</td>
</tr>
<tr>
<td>int</td>
<td>compareTo(Object object)</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(HighResolutionTime time)</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(Object object)</td>
</tr>
<tr>
<td>long</td>
<td>getMilliseconds()</td>
</tr>
<tr>
<td>int</td>
<td>getNanoseconds()</td>
</tr>
<tr>
<td>int</td>
<td>hashCode()</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>relative(Clock clock)</td>
</tr>
<tr>
<td>RelativeTime</td>
<td>relative(Clock clock, HighResolutionTime time)</td>
</tr>
<tr>
<td>void</td>
<td>set(HighResolutionTime time)</td>
</tr>
<tr>
<td>void</td>
<td>set(long millis)</td>
</tr>
<tr>
<td>void</td>
<td>set(long millis, int nanos)</td>
</tr>
<tr>
<td>void</td>
<td>waitForObject(Object target, HighResolutionTime time) throws InterruptedException</td>
</tr>
</tbody>
</table>

## IllegalAssignmentError

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IllegalAssignmentError()</td>
<td></td>
</tr>
<tr>
<td>IllegalAssignmentError(String description)</td>
<td></td>
</tr>
</tbody>
</table>

## ImmortalMemory

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImmortalMemory</td>
<td>instance()</td>
</tr>
</tbody>
</table>
# Chapter 12 Almanac

### ImportanceParameters

```java
package javax.realtime;

public class ImportanceParameters {
    public int getImportance();
    public ImportanceParameters(int priority, int importance);
    public void setImportance(int importance);
    public String toString();
}
```

### InaccessibleAreaException

```java
package javax.realtime;

public class InaccessibleAreaException {
    public InaccessibleAreaException();
    public InaccessibleAreaException(String description);
}
```

### Interruptible

```java
package javax.realtime;

public interface Interruptible {
    public void interruptAction(AsynchronouslyInterruptedException exception);
    public void run(AsynchronouslyInterruptedException exception) throws AsynchronouslyInterruptedException;
}
```

### LTMemory

```java
package javax.realtime;

public class LTMemory {
    public long getMaximumSize();
    public LTMemory(long initialSizeInBytes, long maxSizeInBytes);
    public LTMemory(long initialSizeInBytes, long maxSizeInBytes, Runnable logic);
}
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTMemory(SizeEstimator initial, SizeEstimator maximum)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTMemory(SizeEstimator initial, SizeEstimator maximum, Runnable logic)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String toString()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long size)</td>
<td>SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long base, long size)</td>
<td>SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long base, long size, Runnable logic)</td>
<td>SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long size, Runnable logic)</td>
<td>SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException</td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long base, SizeEstimator size)</td>
<td>SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</td>
<td></td>
</tr>
<tr>
<td>LTPhysicalMemory(Objects type, long base, SizeEstimator size, Runnable logic)</td>
<td>SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</td>
<td></td>
</tr>
</tbody>
</table>
### MemoryAccessError

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryAccessError()</td>
<td></td>
</tr>
<tr>
<td>MemoryAccessError(String description)</td>
<td></td>
</tr>
</tbody>
</table>

### MemoryArea

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryArea()</td>
<td></td>
</tr>
<tr>
<td>MemoryArea(long sizeInBytes)</td>
<td></td>
</tr>
<tr>
<td>MemoryArea(long sizeInBytes, Runnable logic)</td>
<td></td>
</tr>
<tr>
<td>MemoryArea(SizeEstimator size)</td>
<td></td>
</tr>
<tr>
<td>MemoryArea(SizeEstimator size, Runnable logic)</td>
<td></td>
</tr>
<tr>
<td>void enter()</td>
<td>throws ScopedCycleException</td>
</tr>
<tr>
<td>void enter(Runnable logic)</td>
<td>throws ScopedCycleException</td>
</tr>
<tr>
<td>void executeInArea(Runnable logic)</td>
<td>throws InaccessibleAreaException</td>
</tr>
<tr>
<td>long memoryConsumed()</td>
<td></td>
</tr>
<tr>
<td>long memoryRemaining()</td>
<td></td>
</tr>
<tr>
<td>Object newArray(Class type, int number)</td>
<td>throws IllegalAccessException, InstantiationException</td>
</tr>
<tr>
<td>Object newInstance(Class type)</td>
<td>throws IllegalAccessException, InstantiationException</td>
</tr>
<tr>
<td>Object newInstance(reflect.Constructor c, Object[] args)</td>
<td>throws IllegalAccessException, InstantiationException</td>
</tr>
<tr>
<td>long size()</td>
<td></td>
</tr>
</tbody>
</table>

**Object**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throwable</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
</tr>
</tbody>
</table>

**MemoryAccessError**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTPhysicalMemory(Object type, SizeEstimator size)</td>
<td>throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException</td>
</tr>
<tr>
<td>LTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)</td>
<td>throws SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException</td>
</tr>
</tbody>
</table>

**String**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>toString()</td>
<td></td>
</tr>
</tbody>
</table>
MemoryInUseException

<table>
<thead>
<tr>
<th>Object</th>
<th>Throwable java.io.Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exception</td>
</tr>
<tr>
<td></td>
<td>RuntimeException</td>
</tr>
<tr>
<td></td>
<td>MemoryInUseException</td>
</tr>
</tbody>
</table>

MemoryInUseException

MemoryParameters

<table>
<thead>
<tr>
<th>Object</th>
<th>MemoryParameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>getAllocationRate()</td>
</tr>
<tr>
<td>long</td>
<td>getMaxImmortal()</td>
</tr>
<tr>
<td>long</td>
<td>getMaxMemoryArea()</td>
</tr>
<tr>
<td></td>
<td>MemoryParameters(long maxMemoryArea,</td>
</tr>
<tr>
<td></td>
<td>long maxImmortal)</td>
</tr>
<tr>
<td></td>
<td>throws IllegalArgumentException</td>
</tr>
<tr>
<td></td>
<td>MemoryParameters(long maxMemoryArea,</td>
</tr>
<tr>
<td></td>
<td>long maxImmortal, long allocationRate)</td>
</tr>
<tr>
<td></td>
<td>throws IllegalArgumentException</td>
</tr>
<tr>
<td>long</td>
<td>NO_MAX</td>
</tr>
<tr>
<td>void</td>
<td>setAllocationRate(long allocationRate)</td>
</tr>
<tr>
<td>boolean</td>
<td>setAllocationRateIfFeasible(int allocationRate)</td>
</tr>
<tr>
<td>boolean</td>
<td>setMaxImmortalIfFeasible(long maximum)</td>
</tr>
<tr>
<td>boolean</td>
<td>setMaxMemoryAreaIfFeasible(long maximum)</td>
</tr>
</tbody>
</table>

MemoryScopeException

<table>
<thead>
<tr>
<th>Object</th>
<th>Throwable java.io.Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exception</td>
</tr>
<tr>
<td></td>
<td>MemoryScopeException</td>
</tr>
</tbody>
</table>

MemoryScopeException
### MemoryTypeConflictException
```
java.realtime
```
Object
- Throwable
- java.io.Serializable
- Exception
- MemoryTypeConflictException

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryTypeConflictException()</td>
</tr>
<tr>
<td>MemoryTypeConflictException(String s)</td>
</tr>
</tbody>
</table>

### MITViolationException
```
java.realtime
```
Object
- Throwable
- java.io.Serializable
- Exception
- MITViolationException

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MITViolationException()</td>
</tr>
<tr>
<td>MITViolationException(String description)</td>
</tr>
</tbody>
</table>

### MonitorControl
```
java.realtime
```
Object
- MonitorControl

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MonitorControl.getMonitorControl()</td>
</tr>
<tr>
<td>MonitorControl.getMonitorControl(Object monitor)</td>
</tr>
<tr>
<td>MonitorControl.getMonitorControl(Object monitor, MonitorControl monCtl)</td>
</tr>
<tr>
<td>MonitorControl()</td>
</tr>
<tr>
<td>void setMonitorControl(MonitorControl policy)</td>
</tr>
<tr>
<td>void setMonitorControl(Object monitor, MonitorControl monCtl)</td>
</tr>
</tbody>
</table>
### NoHeapRealtimeThread

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
</tbody>
</table>

#### OffsetOutOfBoundsException

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
</tbody>
</table>

#### OneShotTimer

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
<tr>
<td>-</td>
<td><code>null</code></td>
</tr>
</tbody>
</table>

---

class javax.realtime.NoHeapRealtimeThread(SchedulingParameters sp, MemoryArea ma) throws IllegalArgumentException

class javax.realtime.NoHeapRealtimeThread(SchedulingParameters sp, ReleaseParameters rp, MemoryArea ma) throws IllegalArgumentException

class javax.realtime.NoHeapRealtimeThread(SchedulingParameters sp, ReleaseParameters rp, MemoryParameters mp, MemoryArea ma, ProcessingGroupParameters group, Runnable logic) throws IllegalArgumentException

void start()

class javax.realtime.OffsetOutOfBoundsException()

class javax.realtime.OffsetOutOfBoundsException(String description)

class javax.realtime.OneShotTimer(HighResolutionTime start, Clock clock, AsyncEventHandler handler)

class javax.realtime.OneShotTimer(HighResolutionTime time, AsyncEventHandler handler)
CHAPTER 12 ALMANAC

**PeriodicParameters**

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 s)`

**PeriodicTimer**

Object

- AsyncEvent
- Timer
- PeriodicTimer

  ReleaseParameters `createReleaseParameters()`

  void `fire()`

  AbsoluteTime `getFireTime()`

  RelativeTime `getInterval()`

  PeriodicTimer(HighResolutionTime start, RelativeTime interval, AsyncEventHandler handler)

  PeriodicTimer(HighResolutionTime start, RelativeTime interval, Clock clock, AsyncEventHandler handler)

  void `setInterval(RelativeTime interval)`

**PhysicalMemoryManager**

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)
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 getVMAtributes()
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

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
### CHAPTER 12 ALMANAC

<table>
<thead>
<tr>
<th>SIGCONT</th>
<th>int</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGEMT</td>
<td>int</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>int</td>
</tr>
<tr>
<td>SIGFREEZE</td>
<td>int</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>int</td>
</tr>
<tr>
<td>SIGILL</td>
<td>int</td>
</tr>
<tr>
<td>SIGINT</td>
<td>int</td>
</tr>
<tr>
<td>SIGIO</td>
<td>int</td>
</tr>
<tr>
<td>SIGIOT</td>
<td>int</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>int</td>
</tr>
<tr>
<td>SIGLOST</td>
<td>int</td>
</tr>
<tr>
<td>SIGLWP</td>
<td>int</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>int</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>int</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>int</td>
</tr>
<tr>
<td>SIGPWR</td>
<td>int</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>int</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>int</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>int</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>int</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>int</td>
</tr>
<tr>
<td>SIGTHAW</td>
<td>int</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>int</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>int</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>int</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>int</td>
</tr>
<tr>
<td>SIGURG</td>
<td>int</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>int</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>int</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>int</td>
</tr>
<tr>
<td>SIGWAITING</td>
<td>int</td>
</tr>
<tr>
<td>SIGWINCH</td>
<td>int</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>int</td>
</tr>
<tr>
<td>SIGXFSZ</td>
<td>int</td>
</tr>
</tbody>
</table>
 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

  boolean 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()
## ProcessingGroupParameters

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityScheduler instance()</td>
<td>A method to get an instance of the PriorityScheduler class.</td>
</tr>
<tr>
<td>boolean isFeasible()</td>
<td>A method to check if a given schedulable is feasible.</td>
</tr>
<tr>
<td>int MAX_PRIORITY</td>
<td>An integer representing the maximum priority.</td>
</tr>
<tr>
<td>int MIN_PRIORITY</td>
<td>An integer representing the minimum priority.</td>
</tr>
<tr>
<td>PriorityScheduler()</td>
<td>A constructor method for the PriorityScheduler class.</td>
</tr>
<tr>
<td>boolean removeFromFeasibility(Schedulable schedulable)</td>
<td>A method to remove a schedulable from the feasibility list.</td>
</tr>
<tr>
<td>boolean setIfFeasible(Schedulable schedulable,</td>
<td>A method to set if feasible for a given schedulable with release, memory, and processing parameters.</td>
</tr>
<tr>
<td>ReleaseParameters release,</td>
<td></td>
</tr>
<tr>
<td>MemoryParameters memory)</td>
<td></td>
</tr>
<tr>
<td>ProcessingGroupParameters group)</td>
<td></td>
</tr>
<tr>
<td>boolean setIfFeasible(Schedulable schedulable,</td>
<td></td>
</tr>
<tr>
<td>ReleaseParameters release,</td>
<td></td>
</tr>
<tr>
<td>MemoryParameters memory,</td>
<td></td>
</tr>
<tr>
<td>ProcessingGroupParameters group)</td>
<td></td>
</tr>
</tbody>
</table>

### ProcessingGroupParameters

- **javax.realtime**
- **Object**
  - ProcessingGroupParameters
    - RelativeTime getCost()
    - AsyncEventHandler getCostOverrunHandler()
    - RelativeTime getDeadline()
    - AsyncEventHandler getDeadlineMissHandler()
    - RelativeTime getPeriod()
    - HighResolutionTime getStart()

## ProcessingGroupParameters

- 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute(Clock clock, AbsoluteTime destination)</td>
<td>void</td>
</tr>
<tr>
<td>addInterarrivalTo(AbsoluteTime destination)</td>
<td>int</td>
</tr>
<tr>
<td>getInterarrivalTime()</td>
<td>RationalTime(RationalTime dest)</td>
</tr>
<tr>
<td>getInterarrivalTime(RelativeTime dest)</td>
<td>RationalTime(RationalTime dest)</td>
</tr>
</tbody>
</table>

### RawMemoryAccess

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getByte(long offset)</td>
<td>byte</td>
</tr>
<tr>
<td>getBytes(long offset, byte[] bytes, int low, int number)</td>
<td>void</td>
</tr>
<tr>
<td>getInt(long offset)</td>
<td>int</td>
</tr>
<tr>
<td>getInts(long offset, int[] ints, int low, int number)</td>
<td>void</td>
</tr>
<tr>
<td>getLong(long offset)</td>
<td>long</td>
</tr>
<tr>
<td>getLongs(long offset, long[] longs, int low, int number)</td>
<td>void</td>
</tr>
<tr>
<td>getMappedAddress()</td>
<td>long</td>
</tr>
</tbody>
</table>
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)

 mooie[
k* 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

\[
\text{Object RawMemoryAccess RawMemoryFloatAccess}
\]

\[
\text{double getDouble(long offset)}
\]

\[
\text{void getDoubles(long offset, double[]} \text{doubles, int low, int number)}
\]

\[
\text{float getFloat(long offset)}
\]

\[
\text{void getFloats(long offset, float[]} \text{floats, int low, int number)}
\]

\[
\text{RawMemoryFloatAccess(Object type, long size)}
\]

\[
\text{void setDouble(long offset, double value)}
\]

\[
\text{void setDoubles(long offset, double[]} \text{doubles, int low, int number)}
\]

\[
\text{void setFloat(long offset, float value)}
\]

\[
\text{void setFloats(long offset, float[]} \text{floats, int low, int number)}
\]
**Chapter 12 Almanac**

### RealtimeSecurity

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void checkAccessPhysical()</code></td>
<td>Throws SecurityException</td>
</tr>
<tr>
<td><code>void checkAccessPhysicalRange(long base, long size)</code></td>
<td>Throws SecurityException</td>
</tr>
<tr>
<td><code>void checkSetFilter()</code></td>
<td>Throws SecurityException</td>
</tr>
<tr>
<td><code>void checkSetScheduler()</code></td>
<td>Throws SecurityException</td>
</tr>
</tbody>
</table>

### RealtimeSystem

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byte BIG_ENDIAN</code></td>
<td></td>
</tr>
<tr>
<td><code>byte BYTE_ORDER</code></td>
<td></td>
</tr>
<tr>
<td><code>GarbageCollector currentGC()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getConcurrentLocksUsed()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getMaximumConcurrentLocks()</code></td>
<td></td>
</tr>
<tr>
<td><code>RealtimeSecurity getSecurityManager()</code></td>
<td></td>
</tr>
<tr>
<td><code>byte LITTLE_ENDIAN</code></td>
<td></td>
</tr>
</tbody>
</table>

### RealtimeThread

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean addIfFeasible()</code></td>
<td></td>
</tr>
<tr>
<td><code>boolean addToFeasibility()</code></td>
<td></td>
</tr>
<tr>
<td><code>RealtimeThread currentRealtimeThread()</code></td>
<td>Throws ClassCastException</td>
</tr>
<tr>
<td><code>void deschedulePeriodic()</code></td>
<td></td>
</tr>
<tr>
<td><code>MemoryArea getCurrentMemoryArea()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getInitialMemoryAreaIndex()</code></td>
<td></td>
</tr>
<tr>
<td><code>int getMemoryAreaStackDepth()</code></td>
<td></td>
</tr>
</tbody>
</table>
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 memParam)

void setProcessingGroupParameters(ProcessingGroupParameters parameters)

boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)

void setReleaseParameters(ReleaseParameters parameters)

throws IllegalThreadStateException

boolean setReleaseParametersIfFeasible(ReleaseParameters release)

void setScheduler(Scheduler scheduler)

throws IllegalThreadStateException

void setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup)

throws IllegalThreadStateException
Object

- HighResolutionTime
- RelativeTime

RelativeTime

- absolute(Clock clock)
- absolute(Clock clock, AbsoluteTime destination)
- add(long millis, int nanos)
- add(long millis, int nanos, RelativeTime destination)
- add(RelativeTime time)
- add(RelativeTime time, RelativeTime destination)
- addInterarrivalTo(AbsoluteTime destination)
- getInterarrivalTime()
- getInterarrivalTime(RelativeTime destination)
- relative(Clock clock)
- relative(Clock clock, RelativeTime destination)
- RelativeTime()
- RelativeTime(long millis, int nanos)
- RelativeTime(RelativeTime time)
- subtract(RelativeTime time)
- subtract(RelativeTime time, RelativeTime destination)
- toString()

ReleaseParameters

- getCost()
- getCostOverrunHandler()
- getDeadline()
### ResourceLimitError

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResourceLimitError()</td>
<td>Constructor</td>
</tr>
<tr>
<td>ResourceLimitError(String)</td>
<td>Constructor with description</td>
</tr>
</tbody>
</table>

### Schedulable

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToFeasibility()</td>
<td>Add to feasibility</td>
</tr>
<tr>
<td>getMemoryParameters()</td>
<td>Get memory parameters</td>
</tr>
<tr>
<td>getProcessingGroupParameters()</td>
<td>Get processing group parameters</td>
</tr>
<tr>
<td>getReleaseParameters()</td>
<td>Get release parameters</td>
</tr>
<tr>
<td>getScheduler()</td>
<td>Get scheduler</td>
</tr>
<tr>
<td>getSchedulingParameters()</td>
<td>Get scheduling parameters</td>
</tr>
<tr>
<td>removeFromFeasibility()</td>
<td>Remove from feasibility</td>
</tr>
<tr>
<td>setMemoryParameters(MemoryParameters)</td>
<td>Set memory parameters</td>
</tr>
<tr>
<td>setMemoryParametersIfFeasible(MemoryParameters)</td>
<td>Set memory parameters if feasible</td>
</tr>
<tr>
<td>setProcessingGroupParameters(ProcessingGroupParameters)</td>
<td>Set processing group parameters</td>
</tr>
<tr>
<td>setProcessingGroupParametersIfFeasible(ProcessingGroupParameters)</td>
<td>Set processing group parameters if feasible</td>
</tr>
<tr>
<td>setReleaseParameters(ReleaseParameters)</td>
<td>Set release parameters</td>
</tr>
<tr>
<td>setReleaseParametersIfFeasible(ReleaseParameters)</td>
<td>Set release parameters if feasible</td>
</tr>
</tbody>
</table>

### AsyncEventHandler

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDeadlineMissHandler()</td>
<td>Get deadline miss handler</td>
</tr>
<tr>
<td>ReleaseParameters()</td>
<td>Constructor</td>
</tr>
<tr>
<td>ReleaseParameters(RelativeTime, AsyncEventHandler)</td>
<td>Constructor with parameters and handler</td>
</tr>
</tbody>
</table>

### Schedulable Runnable

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setCost(RelativeTime)</td>
<td>Set cost</td>
</tr>
<tr>
<td>setCostOverrunHandler(AsyncEventHandler)</td>
<td>Set cost overrun handler</td>
</tr>
<tr>
<td>setDeadline(RelativeTime)</td>
<td>Set deadline</td>
</tr>
<tr>
<td>setDeadlineMissHandler(AsyncEventHandler)</td>
<td>Set deadline miss handler</td>
</tr>
<tr>
<td>setIfFeasible(RelativeTime, AsyncEventHandler)</td>
<td>Set if feasible</td>
</tr>
</tbody>
</table>
void setScheduler(Scheduler scheduler)
    throws IllegalThreadStateException
void setScheduler(Scheduler scheduler,
    SchedulingParameters scheduling,
    ReleaseParameters release,
    MemoryParameters memoryParameters,
    ProcessingGroupParameters processingGroup)
    throws IllegalThreadStateException
void setSchedulingParameters(SchedulingParameters scheduling)
boolean setSchedulingParametersIfFeasible(SchedulingParameters scheduling)

Scheduler

Object

enum

boolean addToFeasibility(Schedulable schedulable)
void fireSchedulable(Schedulable schedulable)

Scheduler

getDefaultScheduler()

String getPolicyName()

boolean isFeasible()

boolean removeFromFeasibility(Schedulable schedulable)

void setDefaultScheduler(Scheduler scheduler)

boolean setIfFeasible(Schedulable schedulable,
    ReleaseParameters release,
    MemoryParameters memory)

boolean setIfFeasible(Schedulable schedulable,
    ReleaseParameters release,
    MemoryParameters memory,
    ProcessingGroupParameters group)

SchedulingParameters

Object

ScopingCycleException

Object

Throwable java.io.Serializable
ScopedCycleException

ScopedCycleException()

ScopedCycleException(String description)

ScopedMemory

javax.realtime

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, ScopedCycleException

void joinAndEnter(HighResolutionTime time) throws InterruptedException, ScopedCycleException

void joinAndEnter(Runnable logic) throws InterruptedException, ScopedCycleException

void joinAndEnter(Runnable logic, HighResolutionTime time) throws InterruptedException, ScopedCycleException

ScopedMemory(long size)

ScopedMemory(long size, Runnable r)

ScopedMemory(SizeEstimator size)

ScopedMemory(SizeEstimator size, Runnable r)

void setPortal(Object object)

String toString()
### Chapter 12 Almanac

<table>
<thead>
<tr>
<th><strong>SizeEstimator</strong></th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td><strong>SizeEstimator</strong></td>
</tr>
<tr>
<td></td>
<td>long getEstimate()</td>
</tr>
<tr>
<td></td>
<td>void reserve(Class c, int n)</td>
</tr>
<tr>
<td></td>
<td>void reserve(SizeEstimator s)</td>
</tr>
<tr>
<td></td>
<td>void reserve(SizeEstimator s, int n)</td>
</tr>
<tr>
<td></td>
<td>SizeEstimator()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SizeOutOfBoundsException</strong></th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td><strong>Throwable</strong> java.io.Serializable <strong>Exception</strong> SizeOutOfBoundsException</td>
</tr>
<tr>
<td></td>
<td>SizeOutOfBoundsException()</td>
</tr>
<tr>
<td></td>
<td>SizeOutOfBoundsException(String description)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>SporadicParameters</strong></th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td><strong>ReleaseParameters</strong> <strong>AperiodicParameters</strong> <strong>SporadicParameters</strong></td>
</tr>
<tr>
<td></td>
<td>String arrivalTimeQueueOverflowExcept</td>
</tr>
<tr>
<td></td>
<td>String arrivalTimeQueueOverflowIgnore</td>
</tr>
<tr>
<td></td>
<td>String arrivalTimeQueueOverflowReplace</td>
</tr>
<tr>
<td></td>
<td>String arrivalTimeQueueOverflowSave</td>
</tr>
<tr>
<td></td>
<td>String getArrivalTimeQueueOverflowBehavior()</td>
</tr>
<tr>
<td></td>
<td>String getArrivalTimeQueueOverflowBehavior()</td>
</tr>
<tr>
<td></td>
<td>int getInitialArrivalTimeQueueLength()</td>
</tr>
<tr>
<td></td>
<td>int getInitialArrivalTimeQueueLength()</td>
</tr>
<tr>
<td></td>
<td>RelativeTime getMinimumInterarrival()</td>
</tr>
<tr>
<td></td>
<td>String getMitViolationBehavior()</td>
</tr>
<tr>
<td></td>
<td>String getMitViolationBehavior()</td>
</tr>
<tr>
<td></td>
<td>String mitViolationExcept</td>
</tr>
<tr>
<td></td>
<td>String mitViolationIgnore</td>
</tr>
<tr>
<td></td>
<td>String mitViolationReplace</td>
</tr>
<tr>
<td></td>
<td>String mitViolationSave</td>
</tr>
</tbody>
</table>
### ThrowBoundaryError

Object

- Throwable
- Error
- Throwable

- Timer

#### Methods

- **ThrowsBoundaryError()**
- **ThrowsBoundaryError(String description)**

### Timed

Object

- Throwable
- Exception
- InterruptedException
- AsynchronouslyInterruptedException
- Timed

#### Methods

- **doInterruptible(Interruptible logic)**
- **resetTime(HighResolutionTime time)**
- **Timed(HighResolutionTime time)**

#### Throws

- IllegalArgumentException

---

```java
void setArrivalTimeQueueOverflowBehavior(String behavior)
void setArrivalTimeQueueOverflowBehavior(String behavior)
boolean setIfFeasible(RelativeTime interarrival, RelativeTime cost, RelativeTime deadline)
void setInitialArrivalTimeQueueLength(int initial)
void setInitialArrivalTimeQueueLength(int initial)
void setMinimumInterarrival(RelativeTime minimum)
void setMitViolationBehavior(String behavior)
void setMitViolationBehavior(String behavior)

SporadicParameters(RelativeTime minInterarrival, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)

ThrownBoundaryError

javax.realtime

Object

- Throwable
- Error

- ThrowsBoundaryError

#### Methods

- **ThrowsBoundaryError()**
- **ThrowsBoundaryError(String description)**

Timed

javax.realtime

Object

- Throwable
- Exception

- InterruptedException
- AsynchronouslyInterruptedException
- Timed

#### Methods

- **doInterruptible(Interruptible logic)**
- **resetTime(HighResolutionTime time)**
- **Timed(HighResolutionTime time)**

#### Throws

- IllegalArgumentException
```
### CHAPTER 12 Almanac

<table>
<thead>
<tr>
<th>Class</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Timer</strong></td>
<td>javax.realtime</td>
</tr>
<tr>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>➣ AsyncEvent</td>
<td></td>
</tr>
<tr>
<td>➣ Timer</td>
<td></td>
</tr>
<tr>
<td>ReleaseParameters</td>
<td>createReleaseParameters()</td>
</tr>
<tr>
<td>➢ destroy()</td>
<td></td>
</tr>
<tr>
<td>➢ disable()</td>
<td></td>
</tr>
<tr>
<td>➢ enable()</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>getClock()</td>
</tr>
<tr>
<td>AbsoluteTime</td>
<td>getFireTime()</td>
</tr>
<tr>
<td>➢ isRunning()</td>
<td></td>
</tr>
<tr>
<td>➢ reschedule(HighResolutionTime time)</td>
<td></td>
</tr>
<tr>
<td>➢ start()</td>
<td></td>
</tr>
<tr>
<td>➢ stop()</td>
<td></td>
</tr>
<tr>
<td>Timer(HighResolutionTime t, Clock c, AsyncEventHandler handler)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UnknownHappeningException</strong></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>➣ Throwable</td>
<td></td>
</tr>
<tr>
<td>➣ Exception</td>
<td></td>
</tr>
<tr>
<td>➣ RuntimeException</td>
<td></td>
</tr>
<tr>
<td>➣ UnknownHappeningException</td>
<td></td>
</tr>
<tr>
<td>➢ UnknownHappeningException()</td>
<td></td>
</tr>
<tr>
<td>➢ UnknownHappeningException(String description)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UnsupportedPhysicalMemoryException</strong></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td></td>
</tr>
<tr>
<td>➣ Throwable</td>
<td></td>
</tr>
<tr>
<td>➣ Exception</td>
<td></td>
</tr>
<tr>
<td>➣ UnsupportedPhysicalMemoryException</td>
<td></td>
</tr>
<tr>
<td>➢ UnsupportedPhysicalMemoryException()</td>
<td></td>
</tr>
<tr>
<td>➢ UnsupportedPhysicalMemoryException(String description)</td>
<td></td>
</tr>
</tbody>
</table>
VTMemory

java.realtime

Object

- MemoryArea
- ScopedMemory
- VTMemory

<table>
<thead>
<tr>
<th>long getMaximumSize()</th>
</tr>
</thead>
<tbody>
<tr>
<td>String toString()</td>
</tr>
</tbody>
</table>

- VTMemory(long initialSizeInBytes, long maxSizeInBytes)
- VTMemory(long initialSizeInBytes, long maxSizeInBytes, Runnable logic)
- VTMemory(SizeEstimator initial, SizeEstimator maximum)
- VTMemory(SizeEstimator initial, SizeEstimator maximum, Runnable logic)

VTPhysicalMemory

java.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, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
- VTPhysicalMemory(Object type, long base, long size, Runnable logic) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
- VTPhysicalMemory(Object type, long base, long size, Runnable logic) throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException
**WaitFreeDequeue**

<table>
<thead>
<tr>
<th>Method</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>➞ WaitFreeDequeue</td>
<td></td>
</tr>
<tr>
<td>Object blockingRead()</td>
<td></td>
</tr>
<tr>
<td>boolean blockingWrite(Object object)</td>
<td></td>
</tr>
<tr>
<td>throws MemoryScopeException</td>
<td></td>
</tr>
<tr>
<td>boolean force(Object object)</td>
<td></td>
</tr>
<tr>
<td>Object nonBlockingRead()</td>
<td></td>
</tr>
<tr>
<td>boolean nonBlockingWrite(Object object)</td>
<td></td>
</tr>
<tr>
<td>throws MemoryScopeException</td>
<td></td>
</tr>
<tr>
<td><strong>WaitFreeDequeue(Thread writer, Thread reader, int maximum, MemoryArea area)</strong></td>
<td></td>
</tr>
<tr>
<td>throws IllegalArgumentException, IllegalAccessException, IOException</td>
<td></td>
</tr>
</tbody>
</table>

**WaitFreeReadQueue**

<table>
<thead>
<tr>
<th>Method</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>➞ WaitFreeReadQueue</td>
<td></td>
</tr>
<tr>
<td>void clear()</td>
<td></td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td></td>
</tr>
<tr>
<td>boolean isFull()</td>
<td></td>
</tr>
<tr>
<td>Object read()</td>
<td></td>
</tr>
</tbody>
</table>

**VTPhysicalMemory**

<table>
<thead>
<tr>
<th>Method</th>
<th>throws SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>➞ WaitFreeDequeue</td>
<td></td>
</tr>
<tr>
<td>VTPhysicalMemory(Object type, long base, SizeEstimator size)</td>
<td></td>
</tr>
<tr>
<td>VTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic)</td>
<td></td>
</tr>
<tr>
<td>VTPhysicalMemory(Object type, SizeEstimator size)</td>
<td></td>
</tr>
<tr>
<td>VTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)</td>
<td></td>
</tr>
</tbody>
</table>

**WaitFreeDequeue**

<table>
<thead>
<tr>
<th>Method</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>➞ WaitFreeDequeue</td>
<td></td>
</tr>
<tr>
<td>Object blockingRead()</td>
<td></td>
</tr>
<tr>
<td>boolean blockingWrite(Object object)</td>
<td></td>
</tr>
<tr>
<td>throws MemoryScopeException</td>
<td></td>
</tr>
<tr>
<td>boolean force(Object object)</td>
<td></td>
</tr>
<tr>
<td>Object nonBlockingRead()</td>
<td></td>
</tr>
<tr>
<td>boolean nonBlockingWrite(Object object)</td>
<td></td>
</tr>
<tr>
<td>throws MemoryScopeException</td>
<td></td>
</tr>
</tbody>
</table>

**WaitFreeReadQueue**

<table>
<thead>
<tr>
<th>Method</th>
<th>javax.realtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object</strong></td>
<td></td>
</tr>
<tr>
<td>➞ WaitFreeReadQueue</td>
<td></td>
</tr>
<tr>
<td>void clear()</td>
<td></td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td></td>
</tr>
<tr>
<td>boolean isFull()</td>
<td></td>
</tr>
<tr>
<td>Object read()</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>int size()</td>
<td></td>
</tr>
<tr>
<td>void waitForData()</td>
<td></td>
</tr>
<tr>
<td>WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory)</td>
<td>throws IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException</td>
</tr>
<tr>
<td>boolean notify</td>
<td></td>
</tr>
<tr>
<td>WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory, boolean notify)</td>
<td>throws IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException</td>
</tr>
<tr>
<td>boolean write(Object object)</td>
<td>throws MemoryScopeException</td>
</tr>
</tbody>
</table>

**WaitFreeWriteQueue**

```java
Object WaitFreeWriteQueue
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear()</td>
<td></td>
</tr>
<tr>
<td>boolean force(Object object)</td>
<td>throws MemoryScopeException</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td></td>
</tr>
<tr>
<td>boolean isFull()</td>
<td></td>
</tr>
<tr>
<td>Object read()</td>
<td></td>
</tr>
<tr>
<td>int size()</td>
<td></td>
</tr>
<tr>
<td>WaitFreeWriteQueue(Thread writer, Thread reader, int maximum, MemoryArea memory)</td>
<td>throws IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException</td>
</tr>
<tr>
<td>boolean write(Object object)</td>
<td>throws MemoryScopeException</td>
</tr>
</tbody>
</table>
CHAPTER 12 ALMANAC

Obsolete current version is 1.0.1

version is 1.0.1
Bibliography

Wesley, 1996.


BIBLIOGRAPHY


Index

A

absolute(Clock)
of javax.realtime.AbsoluteTime 153
of javax.realtime.HighResolutionTime 149
of javax.realtime.RelativeTime 157

absolute(Clock, AbsoluteTime)
of javax.realtime.AbsoluteTime 153
of javax.realtime.HighResolutionTime 149
of javax.realtime.RationalTime 162
of javax.realtime.RelativeTime 157

AbsoluteTime
of javax.realtime.AbsoluteTime

AbsoluteTime()
of javax.realtime.AbsoluteTime 152

AbsoluteTime(AbsoluteTime)
of javax.realtime.AbsoluteTime 153

AbsoluteTime(Date)
of javax.realtime.AbsoluteTime 153

AbsoluteTime(long, int)
of javax.realtime.AbsoluteTime 153

add(long, int)
of javax.realtime.AbsoluteTime 154
of javax.realtime.RelativeTime 158

add(long, int, AbsoluteTime)
of javax.realtime.AbsoluteTime 154

add(long, int, RelativeTime)
of javax.realtime.RelativeTime 158

add(RelativeTime)
of javax.realtime.AbsoluteTime 154
of javax.realtime.RelativeTime 158

add(RelativeTime, AbsoluteTime)
of javax.realtime.AbsoluteTime 154

add(RelativeTime, RelativeTime)
of javax.realtime.RelativeTime 158

addHandler(AsyncEventHandler)
of javax.realtime.AsyncEvent 181

addHandler(int, AsyncEventHandler)
of javax.realtime.POSIXSignalHandler 208

addIfFeasible()
of javax.realtime.AsyncEventHandler 188
of javax.realtime.RealtimeThread 25

addInterarrivalTo(AbsoluteTime)
of javax.realtime.RationalTime 162
of javax.realtime.RelativeTime 159

addToFeasibility()
of javax.realtime.AsyncEventHandler 188
of javax.realtime.RealtimeThread 25
of javax.realtime.Scheduleable 41

addToFeasibility(Schedulable)
of javax.realtime.PriorityScheduler 48
of javax.realtime.Scheduler 45

ALIGNED
of javax.realtime.PhysicalMemoryManager 95

AperiodicParameters
of javax.realtime 59

AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)
of javax.realtime.AperiodicParameters 60

arrivalTimeQueueOverflowExcept
of javax.realtime.SporadicParameters 62

arrivalTimeQueueOverflowIgnore
of javax.realtime.SporadicParameters 62

arrivalTimeQueueOverflowReplace
of javax.realtime.SporadicParameters 62

arrivalTimeQueueOverflowSave
of javax.realtime.SporadicParameters 63

AsyncEvent
of javax.realtime 181

AsyncEvent()
of javax.realtime.AsyncEvent 181

AsyncEventHandler
of javax.realtime 183

AsyncEventHandler()
of javax.realtime.AsyncEvent 184
INDEX

AsyncEventHandler(boolean)
of javax.realtime.AsyncEventHandler 184
AsyncEventHandler(boolean, Runnable)
of javax.realtime.AsyncEventHandler 185
AsyncEventHandler(Runnable)
of javax.realtime.AsyncEventHandler 185
AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, boolean)
of javax.realtime.AsyncEventHandler 185
AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)
of javax.realtime.AsyncEventHandler 186
AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)
of javax.realtime.AsyncEventHandler 187
AsynchronouslyInterruptedException
of javax.realtime.AsyncEventHandler 198
AsynchronouslyInterruptedException()
of javax.realtime.AsynchronouslyInterruptedException 199

B

BIG_ENDIAN
of javax.realtime.RealtimeSystem 210

BIND TO(String)
of java.realtime.AsyncEvent 181
blockingRead()
of javax.realtime.WaitFreeDequeue 144
blockingWrite(Object)
of javax.realtime.WaitFreeDequeue 145
BoundAsyncEventHandler
of javax.realtime_BOUNDAsyncEventHandler 195

C

checkAccessPhysical()
of javax.realtime.RealtimeSecurity 209
checkAccessPhysicalRange(long, long)
of javax.realtime.RealtimeSecurity 209
checkSetFilter()
of javax.realtime.RealtimeSecurity 209
checkSetScheduler()
of javax.realtime.RealtimeSecurity 210
clear()
of javax.realtime.WaitFreeReadQueue 143
of javax.realtime.WaitFreeWriteQueue 140

Clock
of javax.realtime.Clock 166
Clock()
of javax.realtime.Clock 167
compareTo(HighResolutionTime)
of java.realtime.HighResolutionTime 149
compareTo(Object)
of java.realtime.HighResolutionTime 149
contains(long, long)
of java.realtime.PhysicalMemoryTypeFilter 98
createReleaseParameters()
of java.realtime.AsyncEvent 182
of java.realtime.PeriodicTimer 172
of java.realtime.Timer 169
currentGC()
of javax.realtime.RealtimeSystem 211
currentRealtimeThread()
of javax.realtime.RealtimeThread 25

D
deschedulePeriodic()
of javax.realtime.RealtimeThread 26
destroy()
of javax.realtime.Timer 169
disable()
of javax.realtime.AsynchronouslyInterruptedException 199
of javax.realtime.Timer 169
DMA
of javax.realtime.PhysicalMemoryManager 95
doInterruptible(Interruptible)
of javax.realtime.AsynchronouslyInterruptedException 199
of javax.realtime.Timer 201
DuplicateFilterException
of javax.realtime 214
DuplicateFilterException()
of javax.realtime.DuplicateFilterException 214
DuplicateFilterException(String)
of javax.realtime.DuplicateFilterException 214

E
enable()
of javax.realtime.AsynchronouslyInterruptedException 200
of javax.realtime.Timer 169
enter()
of javax.realtime.MemoryArea 78
of javax.realtime.ScopedMemory 86
enter(Runnable)
of javax.realtime.MemoryArea 78
of javax.realtime.ScopedMemory 86
equals(HighResolutionTime)
of javax.realtime.HighResolutionTime 150
equals(Object)
of javax.realtime.HighResolutionTime 150
executeInArea(Runnable)
of javax.realtime.MemoryArea 79

F
find(long, long)
of javax.realtime.PhysicalMemoryTypeFilter 98
fire()
of javax.realtime.AsynceEvent 182
of javax.realtime.AsynchronouslyInterruptedException 200
of javax.realtime.PeriodicTimer 173
fireSchedulable(Schedulable)
of javax.realtime.PriorityScheduler 48
of javax.realtime.Scheduler 46
force(Object)
of javax.realtime.WaitFreeDequeue 145
of javax.realtime.WaitFreeWriteQueue 140

G
GarbageCollector
of javax.realtime 132
GarbageCollector()
of javax.realtime.GarbageCollector 132
getAllocationRate()
of javax.realtime.MemoryParameters 131
getAndClearPendingFireCount()
of javax.realtime.AsyncEventHandler 188
getAndDecrementPendingFireCount()
of javax.realtime.AsyncEventHandler 188
getAndIncrementPendingFireCount()
of javax.realtime.AsyncEventHandler 189
derivalArrivalTimeQueueOverflowBehavior()
of javax.realtime.SporadicParameters 64
getByte(long)
of javax.realtime.RawMemoryAccess 120
getBytes(long, byte[], int, int)
of javax.realtime.RawMemoryAccess 120
clock()
of javax.realtime.Timer 170
INDEX

getConcurrentLocksUsed()
of javax.realtime.RealtimeSystem 211
getCost()
of javax.realtime.ProcessingGroupParameters 68
of javax.realtime.ReleaseParameters 55
getErrorOverrunHandler()
of javax.realtime.ProcessingGroupParameters 68
of javax.realtime.ReleaseParameters 55
getImportance()
of javax.realtime.ImportanceParameters 53
getInitialArrivalTimeQueueLength()
of javax.realtime.SporadicParameters 65
getInitialMemoryAreaIndex()
of javax.realtime.RealTimeThread 26
getInt(long)
of javax.realtime.RawMemoryAccess 120
getInterarrivalTime()
of javax.realtime.RationalTime 162
of javax.realtime.RelativeTime 159
getInterarrivalTime(RelativeTime)
of javax.realtime.RationalTime 162
of javax.realtime.RelativeTime 159
getInterval()
of javax.realtime.PeriodicTimer 173
getInts(long, int[], int, int)
of javax.realtime.RawMemoryAccess 120
getLong(long)
of javax.realtime.RawMemoryAccess 121
getLongs(long, long[], int, int)
of javax.realtime.RawMemoryAccess 121
getMemoryArea()
of javax.realtime.MemoryParameters 131
getMaximumConcurrentLocks()
of javax.realtime.RealtimeSystem 211
getMaximumSize()
of javax.realtime.LTMemory 94
of javax.realtime.ScopedMemory 87
of javax.realtime.VTMemory 92
getMemoryArea()
of javax.realtime.MemoryParameters 131
getMaxPriority()
of javax.realtime.PriorityScheduler 48
getMaxPriority(Thread)
of javax.realtime.PriorityScheduler 48
getMemoryArea()
of javax.realtime.AsyncEventHandler 189
getMemoryArea(Object)
of javax.realtime.MemoryArea 79
INDEX

H
handleAsyncEvent()
of javax.realtime.AsyncEventHandler 190
handledBy(AsyncEventHandler)
of javax.realtime.AsyncEvent 182
happened(boolean)
of javax.realtime.AsynchronouslyInterruptedException 200
hashCode()
of javax.realtime.HighResolutionTime 150
HeapMemory
of javax.realtime 81
HighResolutionTime
of javax.realtime 148

I
IllegalAssignmentError
of javax.realtime 220
IllegalAssignmentError()
of javax.realtime.IllegalAssignmentError 220
IllegalAssignmentError(String)
of javax.realtime.IllegalAssignmentError 221
ImmortalMemory
of javax.realtime 82
ImmortalPhysicalMemory
of javax.realtime 100
ImmortalPhysicalMemory(Object, long)
of javax.realtime.ImmortalPhysicalMemory 100
ImmortalPhysicalMemory(Object, long, long)
of javax.realtime.ImmortalPhysicalMemory 101
ImmortalPhysicalMemory(Object, long, long, Runnable)
of javax.realtime.ImmortalPhysicalMemory 101
ImmortalPhysicalMemory(Object, long, Runnable)
of javax.realtime.ImmortalPhysicalMemory 102
ImmortalPhysicalMemory(Object, long, SizeEstimator)
of javax.realtime.ImmortalPhysicalMemory 103
ImmortalPhysicalMemory(Object, long, SizeEstimator, Runnable)
of javax.realtime.ImmortalPhysicalMemory 104
ImmortalPhysicalMemory(Object, SizeEstimator)
of javax.realtime.ImmortalPhysicalMemory 104
ImmortalPhysicalMemory(Object, SizeEstimator, Runnable)
of javax.realtime.ImmortalPhysicalMemory 105
ImportanceParameters
of javax.realtime 52
ImportanceParameters(int, int)
of javax.realtime.ImportanceParameters 53
InaccessibleAreaException
of javax.realtime 214
InaccessibleAreaException()
of javax.realtime.InaccessibleAreaException 215
InaccessibleAreaException(String)
of javax.realtime.InaccessibleAreaException 215
initialize(long, long, long)
of javax.realtime.PhysicalMemoryTypeFilter 98
instance()
of javax.realtime.HeapMemory 81
of javax.realtime.ImmortalMemory 82
of javax.realtime.PriorityInheritance 139
of javax.realtime.PriorityScheduler 49
interrupt()
of javax.realtime.RealtimeThread 27
interruptAction(AsynchronouslyInterruptedException)
of javax.realtime.Interruptible 197
Interruptible
of javax.realtime 197
isEmpty()
of javax.realtime.WaitFreeReadQueue 143
of javax.realtime.WaitFreeWriteQueue 140
isEnabled()
  of javax.realtime.AsynchronouslyInterruptedException 200

isFeasible()
  of javax.realtime.PriorityScheduler 49
  of javax.realtime.Scheduler 46

isFull()
  of javax.realtime.WaitFreeReadQueue 143
  of javax.realtime.WaitFreeWriteQueue 140

isPresent(long, long)
  of javax.realtime.PhysicalMemoryTypeFilter 99

isRemovable()
  of javax.realtime.PhysicalMemoryTypeFilter 99

isRemovable(long, long)
  of javax.realtime.PhysicalMemoryManager 96

isRemoved(long, long)
  of javax.realtime.PhysicalMemoryManager 96

isRunning()
  of javax.realtime.Timer 170

J

java.applet - package 223
join()
  of javax.realtime.ScopedMemory 87
join(HighResolutionTime)
  of javax.realtime.ScopedMemory 87
joinAndEnter()
  of javax.realtime.ScopedMemory 88
joinAndEnter(HighResolutionTime)
  of javax.realtime.ScopedMemory 88
joinAndEnter(Runnable)
  of javax.realtime.ScopedMemory 89
joinAndEnter(Runnable, HighResolutionTime)
  of javax.realtime.ScopedMemory 89

L

LITTLE_ENDIAN
  of javax.realtime.RealtimeSystem 210

LTMemory
  of javax.realtime 92
NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryArea)
of javax.realtime.NoHeapRealtimeThread

NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)
of javax.realtime.NoHeapRealtimeThread

nonBlockingRead()
of javax.realtime.WaitFreeDequeue

nonBlockingWrite(Object)
of javax.realtime.WaitFreeDequeue

O

OffsetOutOfBoundsException
of javax.realtime

OffsetOutOfBoundsException()
of javax.realtime.OffsetOutOfBoundsException

OffsetOutOfBoundsException(String)
of javax.realtime.OffsetOutOfBoundsException

OneShotTimer
of javax.realtime

OneShotTimer(HighResolutionTime, AsyncEventHandler)
of javax.realtime.OneShotTimer

OneShotTimer(HighResolutionTime, Clock, AsyncEventHandler)
of javax.realtime.OneShotTimer

onInsertion(long, long, AsyncEventHandler)
of javax.realtime.PhysicalMemoryManager

onRemoval(long, long, AsyncEventHandler)
of javax.realtime.PhysicalMemoryManager

P

PeriodicParameters
of javax.realtime

PeriodicParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)
of javax.realtime.PeriodicParameters

PeriodicTimer
of javax.realtime

PeriodicTimer(HighResolutionTime, RelativeTime, AsyncEventHandler)
of javax.realtime.PeriodicTimer

PeriodicTimer(HighResolutionTime, RelativeTime, Clock, AsyncEventHandler)
of javax.realtime.PeriodicTimer

PhysicalMemoryManager
of javax.realtime

PhysicalMemoryTypeFilter
of javax.realtime

POSIXSignalHandler
of javax.realtime

POSIXSignalHandler()
of javax.realtime.POSIXSignalHandler

PriorityCeilingEmulation
of javax.realtime

PriorityCeilingEmulation(int)
of javax.realtime.PriorityCeilingEmulation

PriorityInheritance
of javax.realtime

PriorityInheritance()
of javax.realtime.PriorityInheritance

PriorityParameters
of javax.realtime

PriorityParameters(int)
of javax.realtime.PriorityParameters

PriorityScheduler
of javax.realtime

PriorityScheduler()
of javax.realtime.PriorityScheduler

ProcessingGroupParameters
of javax.realtime

ProcessingGroupParameters(HighResolutionTime, RelativeTime, RelativeTime, RelativeTime,
AsyncEventHandler, AsyncEventHandler
of javax.realtime.ProcessingGroupParameters 67
propagate()
of javax.realtime.AsynchronouslyInterruptedException 200

R
RationalTime
of javax.realtime 160
RationalTime(int)
of javax.realtime.RationalTime 161
RationalTime(int, long, int)
of javax.realtime.RationalTime 161
RationalTime(int, RelativeTime)
of javax.realtime.RationalTime 161

RawMemoryAccess
of javax.realtime.RawMemoryAccess 117
RawMemoryAccess(Object, long)
of javax.realtime.RawMemoryAccess 118
RawMemoryAccess(Object, long, long)
of javax.realtime.RawMemoryAccess 119

RawMemoryFloatAccess
of javax.realtime.RawMemoryFloatAccess 125
RawMemoryFloatAccess(Object, long)
of javax.realtime.RawMemoryFloatAccess 125
RawMemoryFloatAccess(Object, long, long)
of javax.realtime.RawMemoryFloatAccess 126

read()
of javax.realtime.WaitFreeReadQueue 143
of javax.realtime.WaitFreeWriteQueue 140

RealtimeSecurity
of javax.realtime 209
RealtimeSecurity()
of javax.realtime.RealtimeSecurity 209

RealtimeSystem
of javax.realtime 210
RealtimeSystem()
of javax.realtime.RealtimeSystem 210

RealtimeThread
of javax.realtime 23

RealtimeThread()
of javax.realtime.RealtimeThread 24
RealtimeThread(SchedulingParameters)
of javax.realtime.RealtimeThread 24
RealtimeThread(SchedulingParameters, ReleaseParameters)
of javax.realtime.RealtimeThread 24
RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)
of javax.realtime.RealtimeThread 24
registerFilter(Object, PhysicalMemoryTypeFilter)
of javax.realtime.PhysicalMemoryManager 97
relative(Clock)
of javax.realtime.AbsoluteTime 155
of javax.realtime.HighResolutionTime 150
of javax.realtime.RelativeTime 159
relative(Clock, AbsoluteTime)
of javax.realtime.AbsoluteTime 155
relative(Clock, HighResolutionTime)
of javax.realtime.HighResolutionTime 150
relative(Clock, RelativeTime)
of javax.realtime.RelativeTime 159

RelativeTime
of javax.realtime 156
RelativeTime()
of javax.realtime.RelativeTime 157
RelativeTime(long, int)
of javax.realtime.RelativeTime 157

ReleaseParameters
of javax.realtime 54
ReleaseParameters()
of javax.realtime.ReleaseParameters 54
ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)
of javax.realtime.ReleaseParameters 54
removeFilter(Object)
of javax.realtime.PhysicalMemoryManager 97
removeFromFeasibility()
of javax.realtime.AsyncEventHandler 190
             of javax.realtime.RealtimeThread 27
             of javax.realtime.Schedulable 42
removeFromFeasibility(Schedulable)
of javax.realtime.PriorityScheduler 50
             of javax.realtime.Scheduler 46
removeHandler(AsyncEventHandler)
of javax.realtime.AsyncEvent 182
removeHandler(int, AsyncEventHandler)
of javax.realtime.POSIXSignalHandler 208
reschedule(HighResolutionTime)
of javax.realtime.Timer 170
reserve(Class, int)
of javax.realtime.SizeEstimator 83
reserve(SizeEstimator)
of javax.realtime.SizeEstimator 83
reserve(SizeEstimator, int)
of javax.realtime.SizeEstimator 83
resetTime(HighResolutionTime)
of javax.realtime.Timed 202
ResourceLimitError
         of javax.realtime 221
ResourceLimitError()
of javax.realtime.ResourceLimitError 222
ResourceLimitError(String)
of javax.realtime.ResourceLimitError 222
run()
of javax.realtime.AsyncEventHandler 191
run(AsynchronouslyInterruptedException)
of javax.realtime.Interruptible 197
S
Schedulable
         of javax.realtime 41
schedulePeriodic()
of javax.realtime.RealtimeThread 28
Scheduler
         of javax.realtime 45
Scheduler()
of javax.realtime.Scheduler 45
SchedulingParameters
         of javax.realtime.SchedulingParameters 51
SchedulingParameters()
of javax.realtime.SchedulingParameters 51
ScopedCycleException
         of javax.realtime 219
ScopedCycleException()
of javax.realtime.ScopedCycleException 220
ScopedCycleException(String)
of javax.realtime.ScopedCycleException 220
ScopedMemory
         of javax.realtime 84
ScopedMemory(long)
of javax.realtime.ScopedMemory 85
ScopedMemory(long, Runnable)
of javax.realtime.ScopedMemory 85
ScopedMemory(SizeEstimator)
of javax.realtime.ScopedMemory 85
ScopedMemory(SizeEstimator, Runnable)
of javax.realtime.ScopedMemory 86
set(Date)
of javax.realtime.AbsoluteTime 155
set(HighResolutionTime)
of javax.realtime.HighResolutionTime 151
set(long)
of javax.realtime.HighResolutionTime 151
set(long, int)
of javax.realtime.HighResolutionTime 151
setAllocationRate(long)
of javax.realtime.MemoryParameters 131
setAllocationRateIfFeasible(int)
of javax.realtime.MemoryParameters 131
setArrivalTimeQueueOverflowBehavior(String)
of javax.realtime.SporadicParameters 65
setByte(long, byte)
of javax.realtime.RawMemoryAccess 123
setBytes(long, byte[], int, int)
of javax.realtime.RawMemoryAccess 123
setCost(RelativeTime)
of javax.realtime.ProcessingGroupParameters 69

S
INDEX

of javax.realtime.ReleaseParameters 55
setCostOverrunHandler(AsyncEventHandler)
of javax.realtime.ProcessingGroupParameters 69
of javax.realtime.ReleaseParameters 56
setDeadline(RelativeTime)
of javax.realtime.ProcessingGroupParameters 69
of javax.realtime.ReleaseParameters 56
setDeadlineMissHandler(AsyncEventHandler)
of javax.realtime.ProcessingGroupParameters 69
of javax.realtime.ReleaseParameters 56
setDefaultScheduler(Scheduler)
of javax.realtime.Scheduler 46
setDouble(long, double)
of javax.realtime.RawMemoryFloatAccess 128
setDoubles(long, double[], int, int)
of javax.realtime.RawMemoryFloatAccess 128
setFloat(long, float)
of javax.realtime.RawMemoryFloatAccess 128
setFloats(long, float[], int, int)
of javax.realtime.RawMemoryFloatAccess 129
setFrequency(int)
of javax.realtime.RationalTime 163
setHandler(AsyncEventHandler)
of javax.realtime.AsyncEvent 182
setHandler(int, AsyncEventHandler)
of javax.realtime.POSIXSignalHandler 208
setIfFeasible(RelativeTime, RelativeTime)
of javax.realtime.AperiodicParameters 61
of javax.realtime.ReleaseParameters 57
setIfFeasible(RelativeTime, RelativeTime, RelativeTime)
of javax.realtime.PeriodicParameters 59
of javax.realtime.ReleaseParameters 57
setIfFeasible(RelativeTime, RelativeTime, RelativeTime)
of javax.realtime.PeriodicParameters 59
of javax.realtime.SporadicParameters 66
setIfFeasible(ReleaseParameters, MemoryParameters)
of javax.realtime.AsyncEventHandler 191
of javax.realtime.RealtimeThread 28
setIfFeasible(ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
of javax.realtime.AsyncEventHandler 191
of javax.realtime.RealtimeThread 28
setIfFeasible(ReleaseParameters, ProcessingGroupParameters)
of javax.realtime.AsyncEventHandler 191
of javax.realtime.RealtimeThread 28
setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters)
of javax.realtime.PriorityScheduler 50
of javax.realtime.Scheduler 47
setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
of javax.realtime.PriorityScheduler 50
of javax.realtime.Scheduler 47
setImportance(int)
of javax.realtime.ImportanceParameters 53
setInitialArrivalTimeQueueLength(int)
of javax.realtime.SporadicParameters 66
setInt(long, int)
of javax.realtime.RawMemoryAccess 123
setInterval(RelativeTime)
of javax.realtime.PeriodicTimer 173
setInts(long[], int, int)
of javax.realtime.RawMemoryAccess 123
setLong(long, long)
of javax.realtime.RawMemoryAccess 124
setLongs(long, long[], int, int)
of javax.realtime.RawMemoryAccess 124
setMaxImmortalIfFeasible(long)
of javax.realtime.MemoryParameters 131
setMaximumConcurrentLocks(int)
of javax.realtime.RealtimeSystem 211
setMaximumConcurrentLocks(int, boolean)
of javax.realtime.RealtimeSystem 212
setMaxMemoryAreaIfFeasible(long)
of javax.realtime.MemoryParameters 132
setMemoryParameters(MemoryParameters
Index

- setMemoryParametersIfFeasible(MemoryParameters)
  - of javax.realtime.AsyncEventHandler 192
  - of javax.realtime.RealtimeThread 29
  - of javax.realtime.Schedulable 43

- setMinimumInterarrival(RelativeTime)
  - of javax.realtime.SporadicParameters 66
- setMitViolationBehavior(String)
  - of javax.realtime.SporadicParameters 66
- setMonitorControl(MonitorControl)
  - of javax.realtime.MonitorControl 137
  - of javax.realtime.ScopedMemory 90
- setMonitorControl(Object, MonitorControl)
  - of javax.realtime.MonitorControl 137
- setPeriod(RelativeTime)
  - of javax.realtime.PeriodicParameters 59
  - of javax.realtime.ProcessingGroupParameters 70
- setPortal(Object)
  - of javax.realtime.ScheduledMemory 80
- setPriority(int)
  - of javax.realtime.PriorityParameters 52
- setProcessingGroupParameters(ProcessingGroupParameters)
  - of javax.realtime.AsyncEventHandler 192
  - of javax.realtime.RealtimeThread 29
  - of javax.realtime.Schedulable 43
- setProcessingGroupParametersIfFeasible(ProcessingGroupParameters)
  - of javax.realtime.AsyncEventHandler 192
  - of javax.realtime.RealtimeThread 29
  - of javax.realtime.Schedulable 43
- setReleaseParameters(ReleaseParameters)
  - of javax.realtime.AsyncEventHandler 192
  - of javax.realtime.RealtimeThread 30
  - of javax.realtime.Schedulable 43
- setReleaseParametersIfFeasible(ReleaseParameters)
  - of javax.realtime.AsyncEventHandler 193
  - of javax.realtime.RealtimeThread 30

- setResolution(RelativeTime)
  - of javax.realtime.Clock 168
- setScheduler(Scheduler)
  - of javax.realtime.AsyncEventHandler 193
  - of javax.realtime.RealtimeThread 30
  - of javax.realtime.Schedulable 44
- setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)
  - of javax.realtime.AsyncEventHandler 193
  - of javax.realtime.RealtimeThread 31
  - of javax.realtime.Schedulable 44
- setSchedulingParameters(SchedulingParameters)
  - of javax.realtime.AsyncEventHandler 194
  - of javax.realtime.RealtimeThread 31
  - of javax.realtime.Schedulable 44
- setSchedulingParametersIfFeasible(SchedulingParameters)
  - of javax.realtime.AsyncEventHandler 195
  - of javax.realtime.RealtimeThread 32
  - of javax.realtime.Schedulable 44
- setSecurityManager(RealtimeSecurity)
  - of javax.realtime.RealtimeSystem 212
- setShort(long, short)
  - of javax.realtime.RawMemoryAccess 124
- setShorts(long, short[], int, int)
  - of javax.realtime.RawMemoryAccess 125
- setStart(HighResolutionTime)
  - of javax.realtime.PeriodicParameters 59
  - of javax.realtime.ProcessingGroupParameters 70

- SHARED
  - of javax.realtime.PhysicalMemoryManager 95

- SIGABRT
  - of javax.realtime.POSIXSignalHandler 204

- SIGALRM
  - of javax.realtime.POSIXSignalHandler 204
<table>
<thead>
<tr>
<th>Signal</th>
<th>Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGBUS</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>204</td>
</tr>
<tr>
<td>SIGCANCEL</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>204</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>204</td>
</tr>
<tr>
<td>SIGCLD</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGFREEZE</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGILL</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGINT</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGIO</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGIOT</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGLOST</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>205</td>
</tr>
<tr>
<td>SIGLWP</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGPWR</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGTHAW</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>206</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
<tr>
<td>SIGURG</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>javax.realtime.POSIXSignalHandler</td>
<td>207</td>
</tr>
</tbody>
</table>
SIGUSR2
of javax.realtime.POSIXSignalHandler 207

SIGVTALRM
of javax.realtime.POSIXSignalHandler 207

SIGWINCH
of javax.realtime.POSIXSignalHandler 207

SIGXCPU
of javax.realtime.POSIXSignalHandler 207

SIGXFSZ
of javax.realtime.POSIXSignalHandler 208

size()
of javax.realtime.MemoryArea 81
of javax.realtime.WaitFreeReadQueue 143
of javax.realtime.WaitFreeWriteQueue 141

SizeEstimator
of javax.realtime 82

SizeEstimator()
of javax.realtime.SizeEstimator 83

SizeOutOfBoundsException
of javax.realtime 217

SizeOutOfBoundsException()
of javax.realtime.SizeOutOfBoundsException 218

SizeOutOfBoundsException(String)
of javax.realtime.SizeOutOfBoundsException 218

sleep(Clock, HighResolutionTime)
of javax.realtime.RealtimeThread 32

sleep(HighResolutionTime)
of javax.realtime.RealtimeThread 32

SporadicParameters
of javax.realtime 61

SporadicParameters(RelativeTo, RelativeTime, AsyncEventHandler, AsyncEventHandler)
of javax.realtime.SporadicParameters 63

start()
of javax.realtime.NoHeapRealtimeThread 36

of javax.realtime.RealtimeThread 32
of javax.realtime.Timer 170

stop()
of javax.realtime.Timer 170

subtract(AbsoluteTime)
of javax.realtime.AbsoluteTime 155

subtract(AbsoluteTime, RelativeTime)
of javax.realtime.AbsoluteTime 155

subtract(RelativeTime)
of javax.realtime.AbsoluteTime 156

subtract(RelativeTime, AbsoluteTime)
of javax.realtime.RelativeTime 160

subtract(RelativeTime, RelativeTime)
of javax.realtime.RelativeTime 160

T

ThrowBoundaryError
of javax.realtime 222

ThrowBoundaryError()
of javax.realtime.ThrowBoundaryError 222

ThrowBoundaryError(String)
of javax.realtime.ThrowBoundaryError 222

Timed
of javax.realtime 201

Timed(HighResolutionTime)
of javax.realtime.Timed 201

Timer
of javax.realtime 168

Timer(HighResolutionTime, Clock, AsyncEventHandler)
of javax.realtime.Timed 168

toString()
of javax.realtime.AbsoluteTime 156

of javax.realtime.ImportanceParameters 53

of javax.realtime.LTMemory 94

of javax.realtime.LTPhysicalMemory 111

of javax.realtime.PriorityParameters 52

of javax.realtime.RelativeTime 160

of javax.realtime.ScopedMemory 90

of javax.realtime.VTMemory 92

of javax.realtime.VTPhysicalMemory 117
INDEX

U
unbindTo(String)
of javax.realtime.AsyncEvent 183
UnknownHappeningException
of javax.realtime 220
UnknownHappeningException()
of javax.realtime.UnknownHappeningException 220
UnknownHappeningException(String)
of javax.realtime.UnknownHappeningException 220
unmap()
of javax.realtime.RawMemoryAccess 125
UnsupportedPhysicalMemoryException
of javax.realtime 218
UnsupportedPhysicalMemoryException()
of javax.realtime.UnsupportedPhysicalMemoryException 218
UnsupportedPhysicalMemoryException(String)
of javax.realtime.UnsupportedPhysicalMemoryException 218

V
vFind(long, long)
of javax.realtime.PhysicalMemoryTypeFilter 100
VTMemory
of javax.realtime 90
VTMemory(long, long)
of javax.realtime.VTMemory 91
VTMemory(long, long, Runnable)
of javax.realtime.VTMemory 91
VTMemory(SizeEstimator, SizeEstimator)
of javax.realtime.VTMemory 91
VTMemory(SizeEstimator, SizeEstimator, Runnable)
of javax.realtime.VTMemory 91
VTPhysicalMemory
of javax.realtime 112
VTPhysicalMemory(Object, long)
of javax.realtime.VTPhysicalMemory 112
VTPhysicalMemory(Object, long, long, Runnable)
of javax.realtime.VTPhysicalMemory 113
VTPhysicalMemory(Object, long, Runnable)
of javax.realtime.VTPhysicalMemory 114
VTPhysicalMemory(Object, long, SizeEstimator)
of javax.realtime.VTPhysicalMemory 115
VTPhysicalMemory(Object, long, SizeEstimator, Runnable)
of javax.realtime.VTPhysicalMemory 115
VTPhysicalMemory(Object, SizeEstimator)
of javax.realtime.VTPhysicalMemory 116
VTPhysicalMemory(Object, SizeEstimator, Runnable)
of javax.realtime.VTPhysicalMemory 117

W
waitForData()
of javax.realtime.WaitFreeReadQueue 143
waitForNextPeriod()
of javax.realtime.RealtimeThread 33
waitForObject(Object, HighResolutionTime)
of javax.realtime.HighResolutionTime 151
WaitFreeDequeue
of javax.realtime 144
WaitFreeDequeue(Thread, Thread, int, MemoryArea)
of javax.realtime.WaitFreeDequeue 144
WaitFreeReadQueue
of javax.realtime 141
WaitFreeReadQueue(Thread, Thread, int, MemoryArea)
of javax.realtime.WaitFreeReadQueue 141
WaitFreeReadQueue(Thread, Thread, int, MemoryArea, boolean)
of javax.realtime.WaitFreeReadQueue
INDEX

142
WaitFreeWriteQueue
of javax.realtime 139

WaitFreeWriteQueue(Thread, Thread, int, MemoryArea)
of javax.realtime.WaitFreeWriteQueue
139

write( Object)
of javax.realtime.WaitFreeReadQueue
143
of javax.realtime.WaitFreeWriteQueue
141
Obsolete current version is 1.0.1