# The Real-Time Specification for Java™



## The Real-Time for Java Expert Group

http://www.rtj.org

Greg Bollella

Ben Brosgol
Peter Dibble
Steve Furr
James Gosling
David Hardin
Mark Turnbull

Rudy Belliardi

## The Reference Implementation Team

Doug Locke
Scott Robbins
Pratik Solanki
Dionisio de Niz

**ADDISON-WESLEY**

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

Duke logo designed by Joe Palrang.



The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit Addison-Wesley on the Web at www.awl.com/cseng/

**Library of Congress Control Number: 00-132774**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Contents

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.

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

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

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

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

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

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

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

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

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

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

**Synchronization and Resource Sharing:** Logic often needs to share serializable 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 inverison and allow a thread to have an execution eligibility logically higher than the garbage collector. We provide a set of wait-free queue classes to be used in such situations.

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

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

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

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

Chapter 2

# Design



The RTSJ comprises eight 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:

***Exceptions***

- *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.
- *MemoryAccessError:* 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 ($t_1$,...,$t_{28}$) with priorities ($p_1$,...,$p_{28}$), respectively, where the value of $p_1$ was the highest priority and the value of $p_2$ the next highest priority, etc., then for all executions of threads $t_1$ through $t_{28}$ thread $t_1$ would *always* execute in preference to threads $t_2$, ..., $t_{28}$ and thread $t_2$ would *always* execute in preference to threads $t_3$,..., $t_{28}$, etc.)

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

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

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

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

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

*Optionally Required Components* :

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

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

*Documentation Requirements* :

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

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

*Parameter Objects* :

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

**Caution:** `<string>Parameter` objects are explicitly unsafe in multithreaded situations when they are being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

*Java Platform Dependencies* :

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

1. The `Comparable` interface is available in Java" 2 v1.2 1nd 1.3 and not in what was formally known as JDK’s 1.0 and 1.1. Thus, we expect implementations of this specification which are based on JDK’s 1.0 or 1.1 to include a `Comparable` interface.
2. The class `RawMemoryFloatAccess` is required if and only if the underlying Java Virtual Machine supports floating point data types.

*Illegal Parameter Values* :

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

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

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

Chapter 3

# Threads



This section contains classes that:

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

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

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

## Semantics and Requirements

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

1. The default scheduling policy must manage the execution of instances 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 implmenting the interface `Schedulable` will be the parameter values of the thread invoking the constructor. If the thread invoking the constructor does not have parameter values then the default values are those values associated with the instance of `Scheduler` which will manage the object.
12. Instance 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*

```
public class RealtimeThread extends java.lang.Thread
           implements Schedulable41
```

*All Implemented Interfaces:* java.lang.Runnable, `Schedulable`$_{41}$

*Direct Known Subclasses:* `NoHeapRealtimeThread`$_{33}$

*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`$_{54}$ type of `PeriodicParameters`$_{57}$ , 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`$_{45}$ .

## 3.1.1 Constructors

`public` **`RealtimeThread`**`()`

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

`public` **`RealtimeThread`**`(SchedulingParameters`$_{51}$ `scheduling)`

Create a real-time thread with the given `SchedulingParameters`$_{51}$ .

*Parameters:*

`scheduling` - The `SchedulingParameters`$_{51}$ associated with this (and possibly other `RealtimeThread`).

`public` **`RealtimeThread`**`(SchedulingParameters`$_{51}$ `scheduling,`
`ReleaseParameters`$_{54}$ `release)`

Create a real-time thread with the given `SchedulingParameters`$_{51}$ and `ReleaseParameters`$_{54}$ .

*Parameters:*

`scheduling` - The `SchedulingParameters`$_{51}$ associated with this (and possibly other instances of `RealtimeThread`).

`release` - The `ReleaseParameters`$_{54}$ associated with this (and possibly other instances of `RealtimeThread`).

`public` **`RealtimeThread`**`(SchedulingParameters`$_{51}$ `scheduling,`
`ReleaseParameters`$_{54}$ `release,`
`MemoryParameters`$_{129}$ `memory,`
`MemoryArea`$_{77}$ `area,`

`ProcessingGroupParameters`$_{67}$ `group,`
`java.lang.Runnable logic)`

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

*Parameters:*

`scheduling` - The `SchedulingParameters`$_{51}$ associated with this (and possibly other instances of `RealtimeThread`).

`release` - The `ReleaseParameters`$_{54}$ associated with this (and possibly other instances of `RealtimeThread`).

`memory` - The `MemoryParameters`$_{129}$ associated with this (and possibly other instances of `RealtimeThread`).

`area` - The `MemoryArea`$_{77}$ associated with this.

`group` - The `ProcessingGroupParameters`$_{67}$ 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`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$) of this instance of `Schedulable`$_{41}$ will be considered in the feasibility analysis of the associated `Scheduler`$_{45}$ until further notice. Whether the resulting system is feasible or not, the addition is completed.

*Specified By:* `public boolean addToFeasibility()`$_{41}$ in interface `Schedulable`$_{41}$

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

`public static RealtimeThread`$_{23}$ **`currentRealtimeThread`**`()`
`throws ClassCastException`

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

*Throws:*
`ClassCastException`

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

Stop unblocking `public boolean waitForNextPeriod() throws IllegalThreadStateException`$_{33}$ for a periodic schedulable object. If this does not have a type of `PeriodicParameters`$_{57}$ as it `ReleaseParameters`$_{54}$ nothing happens.

### `public static MemoryArea`$_{77}$ **`getCurrentMemoryArea`**`()`

Return the instance of `MemoryArea`$_{77}$ which is the current memory area for this.

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

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

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

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

Get the size of the stack of `MemoryArea`$_{77}$ instances to which this `RealtimeThread` has access.

*Returns:* The size of the stack of `MemoryArea`$_{77}$ instances.

### `public MemoryParameters`$_{129}$ **`getMemoryParameters`**`()`

Return a reference to the `MemoryParameters`$_{129}$ object.

*Specified By:* `public MemoryParameters`$_{129}$ `getMemoryParameters()`$_{42}$ in interface `Schedulable`$_{41}$

### `public static MemoryArea`$_{77}$ **`getOuterMemoryArea`**`(int index)`

Get the instance of `MemoryArea`$_{77}$ 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`$_{77}$ at index or `null` if the given value is does not correspond to a position in the stack.

`public ProcessingGroupParameters`$_{67}$ **`getProcessingGroupParameters`**`()`

Return a reference to the `ProcessingGroupParameters`$_{67}$ object.

*Specified By:* `public ProcessingGroupParameters`$_{67}$ `getProcessingGroupParameters()`$_{42}$ in interface `Schedulable`$_{41}$

`public ReleaseParameters`$_{54}$ **`getReleaseParameters`**`()`

Returns a reference to the `ReleaseParameters`$_{54}$ object.

*Specified By:* `public ReleaseParameters`$_{54}$ `getReleaseParameters()`$_{42}$ in interface `Schedulable`$_{41}$

`public Scheduler`$_{45}$ **`getScheduler`**`()`

Get the scheduler for this thread.

*Specified By:* `public Scheduler`$_{45}$ `getScheduler()`$_{42}$ in interface `Schedulable`$_{41}$

`public SchedulingParameters`$_{51}$ **`getSchedulingParameters`**`()`

Return a reference to the `SchedulingParameters`$_{51}$ object.

*Specified By:* `public SchedulingParameters`$_{51}$ `getSchedulingParameters()`$_{42}$ in interface `Schedulable`$_{41}$

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

Throw the generic `AsynchronouslyInterruptedException`$_{198}$ at this.

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

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

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$, of this instance of `Schedulable`$_{41}$ should no longer be considered in the feasibility analysis of the associated `Scheduler`$_{45}$. Whether the resulting system is feasible or not, the subtraction is completed.

*Specified By:* `public boolean removeFromFeasibility()`$_{42}$ in interface `Schedulable`$_{41}$

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

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

Begin unblocking `public boolean waitForNextPeriod() throws IllegalThreadStateException`$_{33}$ 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`$_{57}$ as it `ReleaseParameters`$_{54}$ nothing happens.

`public boolean` **`setIfFeasible`**`(ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `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`$_{54}$ `release, MemoryParameters`$_{129}$ `memory, ProcessingGroupParameters`$_{67}$ `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`$_{54}$ `release, ProcessingGroupParameters`$_{67}$ `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`$_{129}$ `parameters)`
`throws IllegalThreadStateException`

Set the reference to the `MemoryParameters`$_{129}$ object.

*Specified By:* `public void setMemoryParameters(MemoryParameters`$_{129}$ `memory)`$_{42}$ in interface `Schedulable`$_{41}$

*Throws:*
`IllegalThreadStateException`

`public boolean` **`setMemoryParametersIfFeasible`**`(MemoryParameters`$_{129}$ `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`$_{129}$ `memParam)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setProcessingGroupParameters`**`(ProcessingGroupParameters`$_{67}$ `parameters)`

Set the reference to the `ProcessingGroupParameters`$_{67}$ object.

*Specified By:* `public void setProcessingGroupParameters(ProcessingGroupParameters`$_{67}$ `groupParameters)`$_{43}$ in interface `Schedulable`$_{41}$

`public boolean` **`setProcessingGroupParametersIfFeasible`**`(Pro` cessingGroupParameters$_{67}$ groupParameters)

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`$_{67}$ `groupParameters)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setReleaseParameters`**`(ReleaseParameters`$_{54}$ `parameters) throws IllegalThreadStateException`

Set the reference to the `ReleaseParameters`$_{54}$ object.

*Specified By:* `public void setReleaseParameters(ReleaseParameters`$_{54}$ `release)`$_{43}$ in interface `Schedulable`$_{41}$

*Throws:*

`IllegalThreadStateException`

`public boolean` **`setReleaseParametersIfFeasible`**`(ReleaseParameters`$_{54}$ `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`$_{54}$ `release)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setScheduler`**`(Scheduler`$_{45}$ `scheduler) throws IllegalThreadStateException`

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

*Specified By:* `public void setScheduler(Scheduler`$_{45}$ `scheduler) throws IllegalThreadStateException`$_{44}$ in interface `Schedulable`$_{41}$

*Throws:*

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

```
public void setScheduler(Scheduler45 scheduler,
        SchedulingParameters51 scheduling,
        ReleaseParameters54 release,
        MemoryParameters129 memoryParameters,
        ProcessingGroupParameters67 processingGroup)
        throws IllegalThreadStateException
```

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

*Specified By:* `public void setScheduler(Scheduler`$_{45}$ `scheduler, SchedulingParameters`$_{51}$ `scheduling, ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `memoryParameters, ProcessingGroupParameters`$_{67}$ `processingGroup) throws IllegalThreadStateException`$_{44}$ in interface `Schedulable`$_{41}$

*Throws:*

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

```
public void setSchedulingParameters(SchedulingParameters51
        scheduling)
        throws IllegalThreadStateException
```

Set the reference to the `SchedulingParameters`$_{51}$ object.

*Specified By:* `public void setSchedulingParameters(SchedulingParameters`$_{51}$ `scheduling)`$_{44}$ in interface `Schedulable`$_{41}$

*Throws:*

`IllegalThreadStateException`

```
public boolean
        setSchedulingParametersIfFeasible(SchedulingParameters[51] 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[51] scheduling)`[44] in interface `Schedulable`[41]

```
public static void sleep(Clock[166] clock,
        HighResolutionTime[148] 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`[166] . 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[148] 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`[166] . 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

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

Used by threads that have a reference to a `ReleaseParameters`54 type of `PeriodicParameters`57 to block until the start of each period. Periods start at either the start time in `PeriodicParameters`57 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`33 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`33 will block until the handler corrects the situation (possibly by calling `public void schedulePeriodic()`28 ). `public boolean waitForNextPeriod() throws IllegalThreadStateException`33 throws `IllegalThreadStateException` if this does not have a reference to a `ReleaseParameters`54 type of `PeriodicParameters`57 .

*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* :

`public class` **`NoHeapRealtimeThread`** `extends RealtimeThread`23

*All Implemented Interfaces:* java.lang.Runnable, `Schedulable`41

*Description* :

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`23 . 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`*84* or `ImmortalMemory`*82* .

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`*45* .

## 3.2.1 Constructors

public **NoHeapRealtimeThread**(`SchedulingParameters`*51* sp,
`MemoryArea`*77* ma)
throws IllegalArgumentException

Create a `NoHeapRealtimeThread`.

*Parameters:*

`scheduling` - A `SchedulingParameters`*51* object that will be associated with this. A null value means this will not have an associated `SchedulingParameters`*51* object.

`area` - A `MemoryArea`*77* object. Must be a `ScopedMemory`*84* or `ImmortalMemory`*82* type. A null value causes an `IllegalArgumentException` to be thrown.

*Throws:*

`IllegalArgumentException`

public **NoHeapRealtimeThread**(`SchedulingParameters`*51* sp,
`ReleaseParameters`*54* rp, `MemoryArea`*77* ma)
throws IllegalArgumentException

Create a `NoHeapRealtimeThread`.

*Parameters:*

`scheduling` - A `SchedulingParameters`*51* object that will be associated with this. A null value means this will not have an associated `SchedulingParameters`*51* object.

`release` - A `ReleaseParameters`*54* object that will be associated with this. A null value means this will not have an associated `ReleaseParameters`*54* object.

`area` - A `MemoryArea`*77* object. Must be a `ScopedMemory`*84* or `ImmortalMemory`*82* type. A null value causes an `IllegalArgumentException` to be thrown.

*Throws:*

`IllegalArgumentException`

```
public NoHeapRealtimeThread(SchedulingParameters51 sp,
            ReleaseParameters54 rp,
            MemoryParameters129 mp, MemoryArea77 ma,
            ProcessingGroupParameters67 group,
            java.lang.Runnable logic)
            throws IllegalArgumentException
```

Create a `NoHeapRealtimeThread`.

*Parameters:*

`scheduling` - A `SchedulingParameters`*51* object that will be associated with this. A null value means this will not have an associated `SchedulingParameters`*51* object.

`release` - A `ReleaseParameters`*54* object that will be associated with this. A null value means this will not have an associated `ReleaseParameters`*54* object.

`memory` - A `MemoryParameters`*129* object that will be associated with this. A null value means this will not have a `MemoryParameters`*129* object.

`area` - A `MemoryArea`*77* object. Must be a `ScopedMemory`*84* or `ImmortalMemory`*82* type. A null value causes an `IllegalArgumentException` to be thrown.

`group` - A `ProcessingGroupParameters`*67* object that will be associated with this. A null value means this will not have an associated `ProcessingGroupParameters`*67* object.

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

*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`$_{221}$

*Overrides:* `public void start()`$_{32}$ in class `RealtimeThread`$_{23}$

*Throws:*

`MemoryAccessError`$_{221}$ - If any of the parameters or `this` is allocated on heap.

Chapter 4

# Scheduling



This section contains classes that:

- Allow the definition of schedulable objects.
- Manage the assignment of execution eligibility to schedulable objects.
- Perform feasibility analysis for sets of schedulable objects.
- Control the admission of new schedulable objects.
- Manage the execution of instances of the `AsyncEventHandler` and `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 SchedulingParamters must be subclassed and the previous statement then holds for the specific subclass type. If, however, additionally provided scheduling policies or execution eligibility assignment policies do not require application visibility of execution eligibility or it appears in another parameter object (e.g., the earliest deadline first scheduling uses deadline as the execution eligibility metric and would thus be visible in ReleaseParameters), then SchedulingParameters need not be subclassed.

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

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

## Semantics and Requirements

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

1. The base scheduler must support at least 28 unique values in the `priorityLevel` field of an instance of `PriorityParameters`.
2. Higher values in the `priorityLevel` field of an instance of `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 thead priorities and the required minimum 28 unique real-time thread priorities shall be from the same space. Assignment of any of these (minimum) 38 priorities to real-time threads or traditional Java threads is legal. It is the responsibility of application logic to make rational priority assignments.
5. The dispatching mechanism must allow the preemption of the execution of schedulable objects at a point not governed by the preempted object.
6. For schedulable objects managed by the base scheduler no part of the system may change the execution eligibility for any reason other than implementation of a priority inversion algorithm. This does not preclude additional schedulers from changing the execution eligibility of schedulable objects—-which they manage—-according to the scheduling algorithm.
7. Threads that are preempted in favor of a higher priority thread may be placed in the appropriate queue at any position as determined by a particular implementation. The implementation is required to provide documentation stating exactly the algorithm used for such placement.
8. If an implementation provides any schedulers other than the base scheduler it shall provide documentation explicitly stating the semantics expressed by 8 through 11 in language and constructs appropriate to the provided scheduling algorithms.
9. All instances of `RelativeTime` used in instances of `ProcessingParameters`, `SchedulingParameters`, and `ReleaseParameters` are measured from the time at which the associated thread (or first such thread) is started.

10. `PriorityScheduler.getNormPriority()` shall be set to `((Priority-Scheduler.getMaxPriority() - PriorityScheduler.getMinPriority())/3) + PriorityScheduler.getMinPriority()`.
11. If instances of `RealtimeThread or NoHeapRealtimeThread` are constructed without a reference to a SchedulingParameters object a `SchedulingParamters` object is created and assigned the values of the current thread. This does not imply that other schedulers should follow this rule. Other schedulers are free to define the default scheduling parameters in the absence of a given `Scheduling-Parameters` object.
12. The policy and semantics embodied in 1 through 11 above and by the descriptions of the refered to classes, methods, and their interactions must be available in all implementations of this specification.
13. This specification does not require any particular feasibility algorithm be implemented in the `Scheduler` object. Those implementations that choose to not implement a feasibility algorithm shall return success whenever the feasibility algorithm is executed.
14. Implementations that provide a scheduler with a feasibility algorithm are required to clearly document the behavior of that algorithm.
15. For instances of `AsyncEventHandler` with a release parameters object of type `SporadicParameters` implementations are required to maintain a list of times at which instances of `AsyncEvent` occurred. The $i^{th}$ time may be removed from the queue after the $i^{th}$ execution of the `handleAsyncEvent` method.
16. If the instance of `AsyncEvent` has more than one instance of `AsyncEvent-Handler` 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 `AsyncEvent-Handler` 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 `MITViolation-Exception` 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* :

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

*All Superinterfaces:* java.lang.Runnable

*All Known Implementing Classes:* `AsyncEventHandler`$_{183}$, `RealtimeThread`$_{23}$

*Description* :

Handlers and other objects can be run by a `Scheduler`$_{45}$ if they provide a `run()` method and the methods defined below. The `Scheduler`$_{45}$ uses this information to create a suitable context to execute the `run()` method.

### 4.1.1 Methods

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

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$) of this instance of `Schedulable`$_{41}$ will be considered in the feasibility analysis of the associated `Scheduler`$_{45}$ until further notice. Whether the resulting system is feasible or not, the addition is completed.

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

`public MemoryParameters`$_{129}$ **`getMemoryParameters`**`()`

Return the `MemoryParameters`$_{129}$ of this schedulable object.

`public ProcessingGroupParameters`$_{67}$ **`getProcessingGroupParameters`**`()`

Return the `ProcessingGroupParameters`$_{67}$ of this schedulable object.

`public ReleaseParameters`$_{54}$ **`getReleaseParameters`**`()`

Return the `ReleaseParameters`$_{54}$ of this schedulable object.

`public Scheduler`$_{45}$ **`getScheduler`**`()`

Return the `Scheduler`$_{45}$ for this schedulable object.

`public SchedulingParameters`$_{51}$ **`getSchedulingParameters`**`()`

Return the `SchedulingParameters`$_{51}$ of this schedulable object.

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

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$, of this instance of `Schedulable`$_{41}$ should no longer be considered in the feasibility analysis of the associated `Scheduler`$_{45}$. Whether the resulting system is feasible or not, the subtraction is completed.

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

`public void` **`setMemoryParameters`**`(MemoryParameters`$_{129}$ `memory)`

Set the `MemoryParameters`$_{129}$ of this schedulable object.

*Parameters:*

`memory` - The `MemoryParameters`$_{129}$ object.

`public boolean` **`setMemoryParametersIfFeasible`**`(MemoryParameters`$_{129}$ `memParam)`

Set the `MemoryParameters`$_{129}$ of this schedulable object.

*Parameters:*

`memory` - The `MemoryParameters`$_{129}$ object. If null nothing happens.

`public void` **`setProcessingGroupParameters`**`(ProcessingGroupParameters`$_{67}$ `groupParameters)`

Set the `ProcessingGroupParameters`$_{67}$ of this schedulable object.

*Parameters:*

`groupParameters` - The `ProcessingGroupParameters`$_{67}$ object.

`public boolean` **`setProcessingGroupParametersIfFeasible`**`(ProcessingGroupParameters`$_{67}$ `groupParameters)`

Set the `ProcessingGroupParameters`$_{67}$ of this schedulable object only if the resulting task set is feasible.

*Parameters:*

`groupParameters` - The `ProcessingGroupParameters`$_{67}$ object.

`public void` **`setReleaseParameters`**`(ReleaseParameters`$_{54}$ `release)`

Set the `ReleaseParameters`$_{54}$ for this schedulable object.

*Parameters:*

`release` - The `ReleaseParameters`$_{54}$ object.

`public boolean` **`setReleaseParametersIfFeasible`**`(ReleaseParameters`$_{54}$ `release)`

Set the `ReleaseParameters`54 for this schedulable object only if the resulting task set is feasible.

*Parameters:*

`release` - The `ReleaseParameters`54 object. If null nothing happens.

```
public void setScheduler(Scheduler45 scheduler)
        throws IllegalThreadStateException
```

Set the `Scheduler`45 for this schedulable object.

*Parameters:*

`scheduler` - The `Scheduler`45 object.

*Throws:*

`IllegalThreadStateException`

```
public void setScheduler(Scheduler45 scheduler,
        SchedulingParameters51 scheduling,
        ReleaseParameters54 release,
        MemoryParameters129 memoryParameters,
        ProcessingGroupParameters67 processingGroup)
        throws IllegalThreadStateException
```

Set the `Scheduler`45 for this schedulable object.

*Parameters:*

`scheduler` - The `Scheduler`45 object.

*Throws:*

`IllegalThreadStateException`

```
public void setSchedulingParameters(SchedulingParameters51
        scheduling)
```

Set the `SchedulingParameters`51 of this scheduable object.

*Parameters:*

`scheduling` - The `SchedulingParameters`51 object.

```
public boolean
        setSchedulingParametersIfFeasible(Scheduli
        ngParameters51 scheduling)
```

Set the `SchedulingParameters`51 of this schedulable object only if the resulting task set is feasible.

*Parameters:*

`scheduling` - The `SchedulingParameters`$_{51}$ object. If null nothing happens.

## 4.2 Scheduler

*Declaration* :

`public abstract class` **`Scheduler`**

*Direct Known Subclasses:* `PriorityScheduler`$_{47}$

*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`$_{45}$ `getDefaultScheduler()`$_{46}$ (e.g., `EDFScheduler`).

### 4.2.1 Constructors

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

Constructor.

### 4.2.2 Methods

`protected abstract boolean` **`addToFeasibility`**`(Schedulable`$_{41}$ `schedulable)`

Inform the scheduler and cooperating facilities that the resource demands (as expressed in the associated instances of `SchedulingParameters`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$) of this instance of `Schedulable`$_{41}$ will be considered in the feasibility analysis of the associated `Scheduler`$_{45}$ 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`$_{41}$ `schedulable)`

Trigger the execution of a schedulable object (like an `AsyncEventHandler`$_{183}$ ).

*Parameters:*

`schedulable` - The schedulable object to make active.

`public static Scheduler`$_{45}$ **`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`$_{41}$ `schedulable)`

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`$_{51}$ , `ReleaseParameters`$_{54}$ , `MemoryParameters`$_{129}$ , and `ProcessingGroupParameters`$_{67}$ , of this instance of `Schedulable`$_{41}$ should no longer be considered in the feasibility analysis of the associated `Scheduler`$_{45}$ . 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`$_{45}$ `scheduler)`

Set the default scheduler. This is the scheduler given to instances of `RealtimeThread`$_{23}$ when they are constructed. The default scheduler is set to the required `PriorityScheduler`$_{47}$ at startup.

*Parameters:*

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

```
public boolean setIfFeasible(Schedulable41 schedulable,
                ReleaseParameters54 release,
                MemoryParameters129 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(Schedulable41 schedulable,
                ReleaseParameters54 release,
                MemoryParameters129 memory,
                ProcessingGroupParameters67 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 Scheduler45`

### 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

protected boolean **addToFeasibility**(Schedulable$_{41}$ schedulable)

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of SchedulingParameters$_{51}$, ReleaseParameters$_{54}$, MemoryParameters$_{129}$, and ProcessingGroupParameters$_{67}$, of this instance of Schedulable$_{41}$ will be considered in the feasibility analysis of the associated Scheduler$_{45}$ until further notice. Whether the resulting system is feasible or not, the addition is completed.

*Overrides:* protected abstract boolean addToFeasibility(Schedulable$_{41}$ schedulable)$_{45}$ in class Scheduler$_{45}$

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

public void **fireSchedulable**(Schedulable$_{41}$ schedulable)

Trigger the execution of a schedulable object (like an instance of AsyncEventHandler$_{183}$).

*Overrides:* public abstract void fireSchedulable(Schedulable$_{41}$ schedulable)$_{46}$ in class Scheduler$_{45}$

*Parameters:*

schedulable - The schedulable object to make active.

public int **getMaxPriority**()

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

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()`46 in class `Scheduler`45

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

`public static PriorityScheduler`47 **`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()`$_{46}$ in class `Scheduler`$_{45}$

### `protected boolean` **`removeFromFeasibility`**`(Schedulable`$_{41}$ `schedulable)`

Inform the scheduler and cooperating facilities that the resource demands, as expressed in the associated instances of `SchedulingParameters`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$, of this instance of `Schedulable`$_{41}$ should no longer be considered in the feasibility analysis of the associated `Scheduler`$_{45}$. Whether the resulting system is feasible or not, the subtraction is completed.

*Overrides:* `protected abstract boolean removeFromFeasibility(Schedulable`$_{41}$ `schedulable)`$_{46}$ in class `Scheduler`$_{45}$

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

### `public boolean` **`setIfFeasible`**`(Schedulable`$_{41}$ `schedulable, ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `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`$_{41}$ `schedulable, ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `memory)`$_{47}$ in class `Scheduler`$_{45}$

### `public boolean` **`setIfFeasible`**`(Schedulable`$_{41}$ `schedulable, ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `memory, ProcessingGroupParameters`$_{67}$ `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`$_{129}$ `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`$_{45}$ . Changes to the values in a parameters object affects the scheduling behavior of all the `Schedulable`$_{41}$ 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

by this specification and represented by the class `PriorityScheduler`47 is such a scheduler.

### 4.5.1 Constructors

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

Create an instance of `SchedulingParameters`51 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 then 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`51

*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`47 is not required to use importance. However, the RTSJ strongly suggests to implementers that a fairly simple subclass of `PriorityScheduler`47 that uses importance can offer value to some real-time applications.

## 4.6.1 Constructors

`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

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

Get the importance value.

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

Set the importance.

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

*Overrides:* `public java.lang.String toString()`52 in class `PriorityParameters`51

## 4.7 ReleaseParameters

*Declaration* :

`public class` **`ReleaseParameters`**

*Direct Known Subclasses:* `AperiodicParameters`$_{59}$, `PeriodicParameters`$_{57}$

*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

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

`protected` **`ReleaseParameters`**`(RelativeTime`$_{156}$ `cost,`
`RelativeTime`$_{156}$ `deadline,`
`AsyncEventHandler`$_{183}$ `overrunHandler,`
`AsyncEventHandler`$_{183}$ `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 `RelativeTime(0,0)` if null.

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

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

`missHandler` - This handler is invoked if the `run()` method of the schedulable object is still executing after the deadline has passed. Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate. If null, nothing happens on the miss deadline condition.

### 4.7.2 Methods

`public RelativeTime`$_{156}$ **`getCost`**`()`

Get the cost value.

`public AsyncEventHandler`$_{183}$ **`getCostOverrunHandler`**`()`

Get the cost overrun handler.

`public RelativeTime`$_{156}$ **`getDeadline`**`()`

Get the deadline.

`public AsyncEventHandler`$_{183}$ **`getDeadlineMissHandler`**`()`

Get the deadline miss handler.

`public void` **`setCost`**`(RelativeTime`$_{156}$ `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 `RelativeTime(0,0)` if null.

`public void` **`setCostOverrunHandler`**`(AsyncEventHandler`*183* `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()`.

`public void` **`setDeadline`**`(RelativeTime`*156* `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.

`public void` **`setDeadlineMissHandler`**`(AsyncEventHandler`*183* `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 deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate.

```
public boolean setIfFeasible(RelativeTime156 cost,
                RelativeTime156 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* :

`public class PeriodicParameters extends ReleaseParameters54`

*Description* :

This release parameter indicates that the `public boolean waitForNextPeriod() throws IllegalThreadStateException33` method on the associated `Schedulable41` 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

```
public PeriodicParameters(HighResolutionTime148 start,
                RelativeTime156 period, RelativeTime156 cost,
                RelativeTime156 deadline,
                AsyncEventHandler183 overrunHandler,
                AsyncEventHandler183 missHandler)
```

Create a `PeriodicParameters` object.

*Parameters:*

`start` - Time at which the first period begins. If a `RelativeTime`156 , 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`152 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`33 . 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

`public RelativeTime`156 **`getPeriod`**`()`

Get the period.

`public HighResolutionTime`$_{148}$ **`getStart`**`()`

Get the start time.

`public boolean` **`setIfFeasible`**`(RelativeTime`$_{156}$ `period,`
`RelativeTime`$_{156}$ `cost,`
`RelativeTime`$_{156}$ `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`$_{156}$ `p)`

Set the period.

*Parameters:*

`period` - The period is the interval between successive unblocks of `public boolean waitForNextPeriod() throws IllegalThreadStateException`$_{33}$. Also used in the feasibility analysis and admission control algorithms.

`public void` **`setStart`**`(HighResolutionTime`$_{148}$ `s)`

Set the start time.

*Parameters:*

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

## 4.9 AperiodicParameters

*Declaration* :
`public class` **`AperiodicParameters`** `extends ReleaseParameters`$_{54}$

*Direct Known Subclasses:* `SporadicParameters`$_{61}$

*Description* :
This release parameter object characterizes a schedulable object that may become active at any time. When a reference to a `AperiodicParameters`$_{59}$ object is given as a parameter to a constructor the `AperiodicParameters`$_{59}$ object becomes bound

to the object being created. Changes to the values in the `AperiodicParameters`$_{59}$ object affect the constructed object. If given to more than one constructor then changes to the values in the `AperiodicParameters`$_{59}$ 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.



## 4.9.1 Constructors

```
public AperiodicParameters(RelativeTime156 cost,
                RelativeTime156 deadline,
                AsyncEventHandler183 overrunHandler,
                AsyncEventHandler183 missHandler)
```

Create an `AperiodicParameters`$_{59}$ object.

*Parameters:*

`cost` - Processing time per invocation. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds cost. Equivalent to `RelativeTime(0,0)` if null.

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. Not used in feasibility analysis for minimum implementation. If null, the deadline will be `RelativeTime(Long.MAX_VALUE,999999)`.

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

`missHandler` - This handler is invoked if the `run()` method of the schedulable object is still executing after the deadline has passed. Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate. If null, nothing happens on the miss deadline condition.

### 4.9.2 Methods

`public boolean` **`setIfFeasible`**`(RelativeTime`$_{156}$ `cost, RelativeTime`$_{156}$ `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`$_{156}$ `cost, RelativeTime`$_{156}$ `deadline)`$_{57}$ in class `ReleaseParameters`$_{54}$

*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* :

`public class` **`SporadicParameters`** `extends AperiodicParameters`$_{59}$

*Description* :

A notice to the scheduler that the associated schedulable object's run method will be released aperiodically but with a minimum time between releases. When a reference to a `SporadicParameters` object is given as a parameter to a constructor, the `SporadicParameters` object becomes bound to the object being created. Changes to the values in the `SporadicParameters` object affect the constructed object. If given to more than one constructor, then changes to the values in the `SporadicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

**Caution:** This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Correct initiation of the deadline miss and cost overrun handlers requires that the underlying system know the arrival time of each sporadic task. For an instance of `RealtimeThread`*23* the arrival time is the time at which the `start()` is invoked. For other instances of `Schedulable`*41* it may be required for the implementation to save the arrival times. For instances of `AsyncEventHandler`*183* with a `ReleaseParameters`*54* 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`*181* bound to the instance of `AsyncEventHandler`*183*.

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

`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`*221*. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler`*183* is bound then the arrival time is ignored.

`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.

`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.

`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`$_{41}$ which has this as its instance of `ReleaseParameters`$_{54}$ occurs at a time less then the minimum interarrival time defined here then the `fire()` method shall throw `MITViolationException`$_{216}$. If the arrival time is a result of a happening to which the instance of `AsyncEventHandler`$_{183}$ is bound then the arrival time is ignored.

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

If an arrival time for any instance of `Schedulable`$_{41}$ which has this as its instance of `ReleaseParameters`$_{54}$ 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`$_{41}$ which has this as its instance of `ReleaseParameters`$_{54}$ 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`$_{41}$ which has this as its instance of `ReleaseParameters`$_{54}$ 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`$_{156}$ `minInterarrival,`
`RelativeTime`$_{156}$ `cost,`

```
RelativeTime156 deadline,
AsyncEventHandler183 overrunHandler,
AsyncEventHandler183 missHandler)
```

Create a `SporadicParameters` object.

*Parameters:*

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

`cost` - Processing time per minimum interarrival interval. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable object receives per interval. On implementations which cannot measure execution time, this value is used as a hint to the feasibility algorithm. On such systems it is not possible to determine when any particular object exceeds cost. Equivalent to `RelativeTime(0,0)` if null.

`deadline` - The latest permissible completion time measured from the release time of the associated invocation of the schedulable object. For a minimum implementation for purposes of feasibility analysis, the deadline is equal to the minimum interarrival interval. Other implementations may use this parameter to compute execution eligibility. If null, deadline will equal the minimum interarrival time.

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

`missHandler` - This handler is invoked if the `run()` method of the schedulable object is still executing after the deadline has passed. Although minimum implementations do not consider deadlines in feasibility calculations, they must recognize variable deadlines and invoke the miss handler as appropriate. If null, nothing happens on the miss deadline condition.

### 4.10.3 Methods

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

Get the behavior of the arrival time queue in the event of an overflow.

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

Get the behavior of the arrival time queue in the event of an overflow.

`public int` **`getInitialArrivalTimeQueueLength`**`()`

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

`public int` **`getInitialArrivalTimeQueueLength`**`()`

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

`public RelativeTime`*156* **`getMinimumInterarrival`**`()`

Get the minimum interarrival time.

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

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

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

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

`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.

`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.

`public boolean` **`setIfFeasible`**`(RelativeTime`[156] `interarrival,`
`RelativeTime`[156] `cost,`
`RelativeTime`[156] `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`[156] `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`$_{59}$ 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(HighResolutionTime148
                start, RelativeTime156 period,
                RelativeTime156 cost,
                RelativeTime156 deadline,
                AsyncEventHandler183 overrunHandler,
                AsyncEventHandler183 missHandler)
```

Create a `ProcessingGroupParameters` object.

*Parameters:*

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

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

`cost` - Processing time per period.

`deadline` - The latest permissible completion time measured from the start of the current period. Changing the deadline might not take effect after the expiration of the current deadline.

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

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

### 4.11.2 Methods

`public RelativeTime`$_{156}$ **`getCost`**`()`

Get the cost value.

`public AsyncEventHandler`$_{183}$ **`getCostOverrunHandler`**`()`

Get the cost overrun handler.

*Returns:* An `AsyncEventHandler`$_{183}$ object that is cost overrun handler of this.

`public RelativeTime`$_{156}$ **`getDeadline`**`()`

Get the deadline value.

*Returns:* A `RelativeTime`$_{156}$ object that represents the deadline of this.

`public AsyncEventHandler`$_{183}$ **`getDeadlineMissHandler`**`()`

Get the deadline missed handler.

*Returns:* An `AsyncEventHandler`$_{183}$ object that is deadline miss handler of this.

`public RelativeTime`$_{156}$ **`getPeriod`**`()`

Get the period.

*Returns:* A `RelativeTime`$_{156}$ object that represents the period of time of this.

public HighResolutionTime$_{148}$ **getStart**()

Get the start time.

*Returns:* A HighResolutionTime$_{148}$ object that represents the start time of this.

public void **setCost**(RelativeTime$_{156}$ cost)

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**(AsyncEventHandler$_{183}$ handler)

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**(RelativeTime$_{156}$ deadline)

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 period.

public void **setDeadlineMissHandler**(AsyncEventHandler$_{183}$ 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.

```
public boolean setIfFeasible(RelativeTime156 period,
                 RelativeTime156 cost,
                 RelativeTime156 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(RelativeTime156 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(HighResolutionTime148 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) <= Cn for some constant C.

2. Execution time of object constructors is explicitly not considered in any bounds.
3. The structure of enclosing scopes is accessible through a set of methods on RealtimeThread. These methods allow the outer scopes to be accessed like an array. The algorithms for maintaining the scope structure are given in “Maintaining the Scope Stack.”
4. A memory scope is represented by an instance of the `ScopedMemory` class. When a new scope is entered, by calling the `enter()` method of the instance or by starting an instance of `RealtimeThread` or `NoHeapRealtimeThread` whose constructors were given a reference to an instance of `ScopedMemory`, all subsequent uses of the `new` keyword within the program logic of the scope will allocate the memory from the memory represented by that instance of `ScopedMemory`. When the scope is exited by returning from the `enter()` method of the instance of `ScopedMemory`, all subsequent uses of the `new` operation will allocate the memory from the area of memory associated with the enclosing scope.
5. The parent of a scoped memory area is the memory area in which the object representing the scoped memory area is allocated.
6. The single parent rule requires that a scope memory area have exactly zero or one parent.
7. Memory scopes that are made current by entering them or passing them as the initial memory area for a new thread must satisfy the *single parent rule.*
8. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count of the number of threads in which it is being used.
9. When the reference count for an instance of the class `ScopedMemory` is decremented from one to zero, all objects within that area are considered unreachable and are candidates for reclamation. The finalizers for each object in the memory associated with an instance of `ScopedMemory` are executed to completion before any statement in any thread attempts to access the memory area again.
10. Objects created in any immortal memory area live for the duration of the application. Their finalizers are only run when the application is terminated.
11. The addresses of objects in any `MemoryArea` that is associated with a `NoHeapRealtimeThread` must remain fixed while they are alive.
12. Each instance of the virtual machine will have exactly one instance of the class `ImmortalMemory`.
13. Each instance of the virtual machine will have exactly one instance of the class `HeapMemory`.

14. Each instance of the virtual machine will behave as if there is an area of memory into which all `Class` objects are placed and which is unexceptionally referenceable by `NoHeapRealtimeThreads`.
15. Strict assignment rules placed on assignments to or from memory areas prevent the creation of dangling pointers, and thus maintain the pointer safety of Java. The restrictions are listed in the following table:

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

16. An implementation must ensure that the above checks are performed on every assignment statement before the statement is executed. (This includes the possibility of static analysis of the application logic).

### Maintaining the Scope Stack

This section describes maintenance of a data structure that is called the *scope stack*. Implementations are not required to use a stack or implement the algorithms given here. It is only required that an implementation behave with respect to the ordering and accessibility of memory scopes effectively as if it implemented these algorithms.

The scope stack is implicitly visible through the assignment rules, and the stack is explicitly visible through the static getOuterMemoryArea(int index) method on RealtimeThread.

Four operations effect the scope stack: the enter method in MemoryArea, construction of a new `RealtimeThread`, the `executeInArea` method in `MemoryArea`, and all the new instance methods in `MemoryArea`.

Notes:

- For the purposes of these algorithms, stacks grow *up*.
- The representative algorithms ignore important issues like freeing objects in scopes.
- In every case, objects in a scoped memory area are eligible to be freed when the reference count for the area goes to zero.

- Any objects in a scoped memory area *must* be freed and their finalizers run before the reference count for the memory area is incremented from zero to one.

**enter**

For `ma.enter(logic)`:

```
if entering ma would violate the single parent rule
    throw ScopedCycleException
push ma on the scope stack belonging to the current thread
execute logic.run method
pop ma from the scope stack
```

**executeInArea or newInstance**

For `ma.executeInArea(logic)`, `ma.newInstance()`, or `ma.newArray()`:

```
if ma is an instance of heap or immortal
    start a new scope stack containing only ma
    make the new scope stack the scope stack for the current threa
d
else ma is scoped
    if ma is in the scope stack for the current thread
        start a new scope stack containing ma and all
              scopes below ma on the scope stack.
        make the new scope stack the scope stack for the current t
hread
    else
        throw InaccessibleAreaException
execute logic.run or construct the object
restore the previous scope stack for the current thread
discard the new scope stack
```

**Construct a RealtimeThread**

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

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

### The Single Parent Rule

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

The parent of a scoped memory area is (for a stack that grows up):

- If the memory area is not currently on any scope stack, it has no parent
- If the memory area is the outermost (lowest) scoped memory area on any scope stack, its parent is the *primordial scope.*
- For all other scoped memory areas, the parent is the first scoped memory are below it on the scope stack.

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

The operational effect of the single parent rule is that once a scoped memory area is assigned a parent none of the above operations can change the parent and thus an ordering imposed by the first assignments of parents of a series of nested scoped memory areas is the only nesting order allowed until control leaves the scopes; then a new nesting order is posible. Thus a thread attempting to enter a scope can only do so by entering in the established nesting order.

### Scope Tree Maintenance

The single parent rule is enforced effectively as if there were a tree with the primordial scope (representing heap and immortal memory) at its root, and other nodes corresponding to ever scoped memory area that is currently on any threads scope stack.

Each scoped memory has a reference to its parent memory area, ma.parent. The parent reference may indicate a specific scoped memory area, no parent, or the primordial parent.

**On Scope Stack Push of ma**

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

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

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

`findFirstScope` is a convenience function that looks down the scope stack for the next entry that is a reference to an instance of `ScopedMemoryArea`.

```
findFirstScope(scopeStack)
   for s = top of scope stack to bottom of scope stack
       if s is an instance of scopedMemory
           return s
   return primordial area
```

**On Scope Stack Pop of ma**

```
ma = t.scopeStack.pop()
if ma is scoped
    ma.refCount--
    if ma.refCount == 0
        ma.parent = noParent
```

## The Rationale

Languages that employ automatic reclamation of blocks of memory allocated in what is traditionally called the heap by program logic also typically use an algorithm called a garbage collector. Garbage collection algorithms and implementations vary in the amount of non-determinancy they add to the execution of program logic. To date, the expert group believes that no garbage collector algorithm or implementation is known that allows preemption at points that leave the inter-object pointers in the heap in a consistent state and are sufficiently close in time to minimize the overhead added to task switch latencies to a sufficiently small enough value which could be considered appropriate for all real-time systems.

Thus, this specification provides the above described areas of memory to allow program logic to allocate objects in a Java-like style, ignore the reclamation of those objects, and not incur the latency of the implemented garbage collection algorithm.

**Illegal Parameters**

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

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

## 5.1 MemoryArea

*Declaration* :

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

*Direct Known Subclasses:* `HeapMemory`81, `ImmortalMemory`82, `ImmortalPhysicalMemory`100, `ScopedMemory`84

*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.

`logic` - The `run()` method of this object will be called whenever
`public void enter()`
`throws ScopedCycleException`$_{78}$ is called.

`protected` **`MemoryArea`**`(SizeEstimator`$_{82}$ `size)`

*Parameters:*

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

`protected` **`MemoryArea`**`(SizeEstimator`$_{82}$ `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`$_{78}$ 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`$_{23}$ or `NoHeapRealtimeThread`$_{33}$.

*Throws:*

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

`ScopedCycleException`$_{219}$ - 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 a `RealtimeThread`$_{23}$ or `NoHeapRealtimeThread`$_{33}$ .

*Parameters:*

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

*Throws:*

`ScopedCycleException`$_{219}$ - If entering this ScopedMemory would violate the single parent rule.

```
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`$_{214}$ - The memory area is not in the thread's scope stack.

```
public static MemoryArea77 getMemoryArea(java.lang.Object
        object)
```

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

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

```
public long memoryConsumed()
```

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

*Returns:* The amount of memory consumed in bytes.

```
public long memoryRemaining()
```

An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

*Returns:* The amount of remaining memory in bytes

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

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.

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

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.

`OutOfMemoryError` - Space in the memory area is exhausted.

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

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.

`OutOfMemoryError` - Space in the memory area is exhausted.

```
public long size()
```

Query the size of the memory area. The returned value is the current size. Current size may be larger than initial size for those areas that are allowed to grow.

*Returns:* The size of the memory area in bytes.

## 5.2 HeapMemory

*Declaration* :

`public final class HeapMemory extends MemoryArea`77

*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

```
public static HeapMemory81 instance()
```

Returns a pointer to the singleton `HeapMemory` space.

*Returns:* The singleton `HeapMemory` object.

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

*Overrides:* `public long memoryConsumed()`$_{79}$ in class `MemoryArea`$_{77}$

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

*Overrides:* `public long memoryRemaining()`$_{80}$ in class `MemoryArea`$_{77}$

## 5.3 ImmortalMemory

*Declaration* :

`public final class` **`ImmortalMemory`** `extends MemoryArea`$_{77}$

*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

`public static ImmortalMemory`$_{82}$ **`instance`**`()`

Returns a pointer to the singleton `ImmortalMemory` space.

*Returns:* The singleton `ImmortalMemory` object.

## 5.4 SizeEstimator

*Declaration* :

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

*Description* :

This is a convenient class to help people figure out how much memory they need. Instead of passing actual numbers to the `MemoryArea` 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`$_{82}$ `size)`$_{78}$, `public LTMemory(SizeEstimator`$_{82}$ `initial, SizeEstimator`$_{82}$ `maximum)`$_{94}$, `public VTMemory(SizeEstimator`$_{82}$ `initial, SizeEstimator`$_{82}$ `maximum)`$_{91}$

## 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`$_{77}$.

*Parameters:*

`c` - The class to take into account.

`n` - The number of instances of `c` to estimate.

`public void` **`reserve`**`(SizeEstimator`$_{82}$ `s)`

Take into account an additional instance of SizeEstimator `s` when estimating the size of the `MemoryArea`$_{77}$.

*Parameters:*

`c` - The class to take into account.

`n` - The number of instances of `c` to estimate.

`public void` **`reserve`**`(SizeEstimator`$_{82}$ `s, int n)`

Take into account additional `n` instances of SizeEstimator `s` when estimating the size of the `MemoryArea`$_{77}$.

*Parameters:*

`c` - The class to take into account.

`n` - The number of instances of `c` to estimate.

---

## 5.5 ScopedMemory

*Declaration* :

`public abstract class` **`ScopedMemory`** `extends MemoryArea`$_{77}$

*Direct Known Subclasses:* `LTMemory`$_{92}$, `LTPhysicalMemory`$_{106}$, `VTMemory`$_{90}$, `VTPhysicalMemory`$_{112}$

*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() throws ScopedCycleException`$_{86}$ 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:

```
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`$_{86}$

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`*82* .
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

`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.

`public` **`ScopedMemory`**`(long size, java.lang.Runnable 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.lang.Runnable` whose `run()` method is invoked when any of the variations of enter() which do not take a `java.lang.Runnable` is called.

`public` **`ScopedMemory`**`(SizeEstimator`*82* `size)`

Create a new `ScopedMemory` with size equal to `size.getEstimate()`.

*Parameters:*

`size` - A (@link SizeEstimator} which encapsulates the size of the new `ScopedMemory` area.

```
public ScopedMemory(SizeEstimator82 size,
            java.lang.Runnable r)
```

Create a new `ScopedMemory` with size equal to `size.getEstimate()`. 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

```
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`$_{23}$ or `NoHeapRealtimeThread`$_{33}$.

*Overrides:* `public void enter() throws ScopedCycleException`$_{78}$ in class `MemoryArea`$_{77}$

*Parameters:*

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

*Throws:*

`ScopedCycleException`$_{219}$

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

Associate this `ScopedMemory` area to the current realtime thread for the duration of the execution of the `run()` method of the given `java.lang.Runnable`. During this bound period of execution, all objects are allocated from the `ScopedMemory` area until another one takes effect, or the `enter()` method is exited. A runtime exception is thrown if this method is called from a thread other than a `RealtimeThread`$_{23}$ or `NoHeapRealtimeThread`$_{33}$.

*Overrides:* `public void enter(java.lang.Runnable logic) throws ScopedCycleException`$_{78}$ in class `MemoryArea`$_{77}$

*Parameters:*

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

*Throws:*

`ScopedCycleException`$_{219}$

`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)`$_{90}$

*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`$_{148}$ `time)`

```
throws InterruptedException
```

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

*Parameters:*

`time` - If this time is an absolute time, the wait is bounded by that point in time. If the time is a relative time (or a member of the `RationalTime` subclass of `RelativeTime` the wait is bounded by a the specified interval from some time between the time `join` is called and the time it starts waiting for the reference count to reach zero.

*Throws:*

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

```
public void joinAndEnter()
        throws InterruptedException, ScopedCycleEx
        ception
```

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

```
public void joinAndEnter(HighResolutionTime148 time)
        throws InterruptedException, ScopedCycleEx
        ception
```

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`[219] - If entering this `ScopedMemory` would violate the single parent rule.

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

Combine `join();enter(logic);` such that no `enter` from another thread can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, then enter the `ScopedMemory` and execute the `run` method from `logic`

*Parameters:*

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

*Throws:*

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

`ScopedCycleException`[219] - If entering this `ScopedMemory` would violate the single parent rule.

```
public void joinAndEnter(java.lang.Runnable logic,
        HighResolutionTime[148] time)
        throws InterruptedException, ScopedCycleEx
        ception
```

Combine `join(time);enter(logic);` such that no `enter` from another thread can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, or for the current time to reach the designated time, then enter the `ScopedMemory` and execute the `run` method from `logic`.

*Parameters:*

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

`time` - The time that bounds the wait.

*Throws:*

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

`ScopedCycleException`$_{219}$ - 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`$_{84}$

*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`$_{33}$.

### 5.6.1 Constructors

```
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.
- `maximumSizeInBytes` - The maximum size in bytes this memory area can grow to.

```
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.
- `maximumSizeInBytes` - The maximum size in bytes this memory area can grow to.
- `logic` - The logic associated with this.

public **VTMemory**(`SizeEstimator`$_{82}$ initial,
`SizeEstimator`$_{82}$ maximum)

Creates a `VTMemory` of the given size estimated by two instances of `SizeEstimator`$_{82}$.

*Parameters:*

- `initial` - The instance of `SizeEstimator`$_{82}$ which will set the initial allocation allocate for this area.
- `maximum` - The instance of `SizeEstimator`$_{82}$ which will set the maximum allocation allocate for this area.

public **VTMemory**(`SizeEstimator`$_{82}$ initial,
`SizeEstimator`$_{82}$ maximum,
java.lang.Runnable logic)

Creates a `VTMemory` of the given size estimated by two instances of `SizeEstimator`$_{82}$ and logic.

*Parameters:*

- `initial` - The instance of `SizeEstimator`$_{82}$ which will set the initial allocation allocate for this area.
- `maximum` - The instance of `SizeEstimator`$_{82}$ which will set the maximum allocation allocate 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()`$_{87}$ in class `ScopedMemory`$_{84}$

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

*Overrides:* `public java.lang.String toString()`$_{90}$ in class `ScopedMemory`$_{84}$

---

## 5.7 LTMemory

*Declaration* :

`public class` **`LTMemory`** `extends ScopedMemory`$_{84}$

*Description* :

`LTMemory` represents a memory area, allocated per `RealtimeThread`$_{23}$, 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`$_{33}$, or to enter the memory area using the `public void enter() throws ScopedCycleException`$_{86}$ method within a `NoHeapRealtimeThread`$_{33}$. An `LTMemory` area has an initial size. Enough memory must be committed by the completion of the constructor to satisfy this initial requirement. (Committed means that this memory must always be available for allocation). The initial memory allocation must behave, with respect to successful allocation, as if it were contiguous; i.e., a correct implementation must guarantee that any sequence of object allocations that could ever succeed without exceeding a specified initial memory size will always succeed without exceeding that initial memory size and succeed for any instance of `LTMemory` with that initial memory size. *(Note: It is important to understand that the above statement does* ***not require that if the initial memory size is N and (sizeof(object1) + sizeof(object2) + ... + 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. (Note: to ensure that all requested memory is available set initial and maximum to the same value)

*See Also:* `MemoryArea`*77*, `ScopedMemory`*84*, `RealtimeThread`*23*, `NoHeapRealtimeThread`*33*

## 5.7.1 Constructors

```
public LTMemory(long initialSizeInBytes,
                long maxSizeInBytes)
```

Create an `LTMemory` of the given size.

*Parameters:*

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

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

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

Create an `LTMemory` of the given size and logic.

*Parameters:*

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

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

`public LTMemory(SizeEstimator82 initial, SizeEstimator82 maximum)`

Creates a `LTMemory` of the given size estimated by two instances of `SizeEstimator`$_{82}$.

*Parameters:*

`initial` - The instance of `SizeEstimator`$_{82}$ which will set the initial allocation allocate for this area.

`maximum` - The instance of `SizeEstimator`$_{82}$ which will set the maximum allocation allocate for this area.

`public LTMemory(SizeEstimator82 initial, SizeEstimator82 maximum, java.lang.Runnable logic)`

Creates a `LTMemory` of the given size estimated by two instances of `SizeEstimator`$_{82}$ and logic.

*Parameters:*

`initial` - The instance of `SizeEstimator`$_{82}$ which will set the initial allocation allocate for this area.

`maximum` - The instance of `SizeEstimator`$_{82}$ which will set the maximum allocation allocate for this area.

`logic` - The logic associated with this.

## 5.7.2 Methods

`public long getMaximumSize()`

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

*Overrides:* `public long getMaximumSize()`$_{87}$ in class `ScopedMemory`$_{84}$

`public java.lang.String toString()`

*Overrides:* `public java.lang.String toString()`$_{90}$ in class `ScopedMemory`$_{84}$

## 5.8 PhysicalMemoryManager

*Declaration* :

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

*Description* :

The `PhysicalMemoryManager` is available for use by the various physical memory accessor objects (`VTPhysicalMemory`$_{112}$, `LTPhysicalMemory`$_{106}$, `ImmortalPhysicalMemory`$_{100}$, `RawMemoryAccess`$_{117}$, and `RawMemoryFloatAccess`$_{125}$) 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`$_{98}$ classes that correctly identify memory classes that are standard for the (OS, JVM, and processor) platform.

OEMs may provide `PhysicalMemoryTypeFilter`$_{98}$ 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`$_{215}$, but since filters are not part of the normative RTSJ, this exception is at best advisory.

### 5.8.1 Fields

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

Specify this to identify aligned memory.

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

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

`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

`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

`public static void` **`onInsertion`**`(long base, long size, AsyncEventHandler`*183*` aeh)`

Register the specified `AsyncEventHandler`*183* to run when any memory in the range is added to the system. If the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with any one of them. If the size or the base is less than 0, unregister all “remove” references to the 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.

`public static void` **`onRemoval`**`(long base, long size, AsyncEventHandler`*183*` 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.

```
public static final void registerFilter(java.lang.Object
        name, PhysicalMemoryTypeFilter98 filter)
        throws DuplicateFilterException, IllegalAr
        gumentException
```

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`214 - 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.

```
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.

## 5.9 PhysicalMemoryTypeFilter

*Declaration* :

`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`$_{183}$ `aeh)`

Arrange for the specified `AsyncEventHandler`$_{183}$ 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`$_{183}$ `aeh)`

Arrange for the specified `AsyncEventHandler`$_{183}$ 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`77

*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`82 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, SizeOutOfBoundsE
               xception, UnsupportedPhysicalMemoryExcepti
               on, 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`*217* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*218* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*215*

```
public ImmortalPhysicalMemory(java.lang.Object type,
            long base, long size)
            throws SecurityException, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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` - 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`*217* - The address is invalid.

`SizeOutOfBoundsException`*217* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*218* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*215* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`*219* - The specified memory is already in use.

```
public ImmortalPhysicalMemory(java.lang.Object type,
            long base, long size,
```

```
java.lang.Runnable logic)
throws SecurityException, SizeOutOfBoundsE
xception, OffsetOutOfBoundsException, Unsu
pportedPhysicalMemoryException, 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` - The size of the area in bytes.

`logic` - The `run()` method of this object will be called whenever `public void enter() throws ScopedCycleException`$_{78}$ is called.

*Throws:*

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

`OffsetOutOfBoundsException`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public ImmortalPhysicalMemory(java.lang.Object type,
            long size, java.lang.Runnable logic)
            throws SecurityException, SizeOutOfBoundsE
            xception, UnsupportedPhysicalMemoryExcepti
            on, 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` - The `run()` method of this object will be called whenever
`public void enter()`
`throws ScopedCycleException`78 is called.

*Throws:*

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

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

`UnsupportedPhysicalMemoryException`218 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`215 - 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, SizeEstimator82 size)
            throws SecurityException, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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`217 - The address is invalid.

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

`UnsupportedPhysicalMemoryException`218 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`215 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public ImmortalPhysicalMemory(java.lang.Object type,
             long base, SizeEstimator82 size,
             java.lang.Runnable logic)
             throws SecurityException, SizeOutOfBoundsE
             xception, OffsetOutOfBoundsException, Unsu
             pportedPhysicalMemoryException, 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.

`logic` - The `run()` method of this object will be called whenever `public void enter() throws ScopedCycleException`$_{78}$ is called.

*Throws:*

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

`OffsetOutOfBoundsException`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public ImmortalPhysicalMemory(java.lang.Object type,
             SizeEstimator82 size)
             throws SecurityException, SizeOutOfBoundsE
             xception, UnsupportedPhysicalMemoryExcepti
             on, 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`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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,
            SizeEstimator82 size,
            java.lang.Runnable logic)
            throws SecurityException, SizeOutOfBoundsE
            xception, UnsupportedPhysicalMemoryExcepti
            on, 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`$_{78}$ is called.

*Throws:*

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

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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* :

`public class` **`LTPhysicalMemory`** `extends ScopedMemory`$_{84}$

*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`$_{84}$ memory areas, and the same performance restrictions as `LTMemory`.

*See Also:* `MemoryArea`$_{77}$, `ScopedMemory`$_{84}$, `VTMemory`$_{90}$, `LTMemory`$_{92}$, `VTPhysicalMemory`$_{112}$, `ImmortalPhysicalMemory`$_{100}$, `RealtimeThread`$_{23}$, `NoHeapRealtimeThread`$_{33}$

### 5.11.1 Constructors

```
public LTPhysicalMemory(java.lang.Object type, long size)
          throws SecurityException, SizeOutOfBoundsE
          xception, UnsupportedPhysicalMemoryExcepti
          on, 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`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`215 - 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, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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` - 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`217 - The address is invalid.

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

`UnsupportedPhysicalMemoryException`218 - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`215 - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`219 - The specified memory is already in use.

```
public LTPhysicalMemory(java.lang.Object type, long base,
            long size, java.lang.Runnable logic)
            throws SecurityException, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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` - 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`*217* - The address is invalid.

`SizeOutOfBoundsException`*217* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*218* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*215* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`*219* - The specified memory is already in use.

```
public LTPhysicalMemory(java.lang.Object type, long size,
            java.lang.Runnable logic)
            throws SecurityException, SizeOutOfBoundsE
            xception, UnsupportedPhysicalMemoryExcepti
            on, 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.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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,
          SizeEstimator82 size)
          throws SecurityException, SizeOutOfBoundsE
          xception, OffsetOutOfBoundsException, Unsu
          pportedPhysicalMemoryException, 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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public LTPhysicalMemory(java.lang.Object type, long base,
          SizeEstimator82 size,
          java.lang.Runnable logic)
          throws SecurityException, SizeOutOfBoundsE
```

```
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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.

`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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public LTPhysicalMemory(java.lang.Object type,
            SizeEstimator82 size)
            throws SecurityException, SizeOutOfBoundsE
            xception, UnsupportedPhysicalMemoryExcepti
            on, 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`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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,
        SizeEstimator82 size,
        java.lang.Runnable logic)
        throws SecurityException, SizeOutOfBoundsE
        xception, UnsupportedPhysicalMemoryExcepti
        on, 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`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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()`$_{90}$ in class `ScopedMemory`$_{84}$

## 5.12 VTPhysicalMemory

*Declaration* :

`public class **VTPhysicalMemory** extends ScopedMemory`$_{84}$

*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`$_{84}$ memory areas, and the same performance restrictions as `VTMemory`.

*See Also:* `MemoryArea`$_{77}$, `ScopedMemory`$_{84}$, `VTMemory`$_{90}$, `LTMemory`$_{92}$, `LTPhysicalMemory`$_{106}$, `ImmortalPhysicalMemory`$_{100}$, `RealtimeThread`$_{23}$, `NoHeapRealtimeThread`$_{33}$

### 5.12.1 Constructors

```
public VTPhysicalMemory(java.lang.Object type, long size)
                 throws SecurityException, SizeOutOfBoundsE
                 xception, UnsupportedPhysicalMemoryExcepti
                 on, 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`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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,
```

```
                  long size)
                  throws SecurityException, SizeOutOfBoundsE
                  xception, OffsetOutOfBoundsException, Unsu
                  pportedPhysicalMemoryException, 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` - 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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public VTPhysicalMemory(java.lang.Object type, long base,
                  long size, java.lang.Runnable logic)
                  throws SecurityException, SizeOutOfBoundsE
                  xception, OffsetOutOfBoundsException, Unsu
                  pportedPhysicalMemoryException, 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` - 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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public VTPhysicalMemory(java.lang.Object type, long size,
        java.lang.Runnable logic)
        throws SecurityException, SizeOutOfBoundsE
        xception, UnsupportedPhysicalMemoryExcepti
        on, 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.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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,
            SizeEstimator82 size)
            throws SecurityException, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public VTPhysicalMemory(java.lang.Object type, long base,
            SizeEstimator82 size,
            java.lang.Runnable logic)
            throws SecurityException, SizeOutOfBoundsE
            xception, OffsetOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, 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.

`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`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

`MemoryInUseException`$_{219}$ - The specified memory is already in use.

```
public VTPhysicalMemory(java.lang.Object type,
            SizeEstimator82 size)
            throws SecurityException, SizeOutOfBoundsE
            xception, UnsupportedPhysicalMemoryExcepti
            on, 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`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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`$_{82}$ `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`$_{217}$ - The size extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$ - 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()`$_{90}$ in class `ScopedMemory`$_{84}$

## 5.13 RawMemoryAccess

*Declaration* :

`public class` **`RawMemoryAccess`**

*Direct Known Subclasses:* `RawMemoryFloatAccess`$_{125}$

*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`$_{210}$ .

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`$_{217}$ . 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`$_{217}$ . 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

```
public RawMemoryAccess(java.lang.Object type, long size)
            throws SecurityException, OffsetOutOfBound
            sException, SizeOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, MemoryType
            ConflictException
```

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`*217* - The address is invalid.

`SizeOutOfBoundsException`*217* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*218* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*215* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

```
public RawMemoryAccess(java.lang.Object type, long base,
          long size)
          throws SecurityException, OffsetOutOfBound
          sException, SizeOutOfBoundsException, Unsu
          pportedPhysicalMemoryException, MemoryType
          ConflictException
```

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`*217* - The address is invalid.

`SizeOutOfBoundsException`*217* - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`*218* - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`*215* - The specified base does not point to memory that matches the request type, or if `type` specifies attributes with a conflict.

### 5.13.2 Methods

```
public byte getByte(long offset)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Get the `byte` at the given offset.

*Parameters:*

`offset` - The offset at which to read the byte.

*Returns:* The byte read.

*Throws:*

`SizeOutOfBoundsException`$_{217}$,
`OffsetOutOfBoundsException`$_{217}$

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

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

*Throws:*

`SizeOutOfBoundsException`$_{217}$,
`OffsetOutOfBoundsException`$_{217}$

```
public int getInt(long offset)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Get the `int` at the given offset.

*Parameters:*

`offset` - The offset at which to read the integer.

*Returns:* The int read.

*Throws:*

`SizeOutOfBoundsException`$_{217}$,
`OffsetOutOfBoundsException`$_{217}$

```
public void getInts(long offset, int[] ints, int low,
            int number)
```

```
throws OffsetOutOfBoundsException, SizeOut
OfBoundsException
```

Get `number` ints starting at the given offset and assign them to the int array passed starting at position `low`.

*Throws:*

`SizeOutOfBoundsException`217, `OffsetOutOfBoundsException`217

```
public long getLong(long offset)
        throws OffsetOutOfBoundsException, SizeOut
        OfBoundsException
```

Get the `long` at the given offset.

*Parameters:*

`offset` - The offset at which to read the long.

*Returns:* The long read.

*Throws:*

`SizeOutOfBoundsException`217, `OffsetOutOfBoundsException`217

```
public void getLongs(long offset, long[] longs, int low,
        int number)
        throws OffsetOutOfBoundsException, SizeOut
        OfBoundsException
```

Get `number` longs starting at the given offset and assign them to the long array passed starting at position `low`.

*Throws:*

`SizeOutOfBoundsException`217, `OffsetOutOfBoundsException`217

```
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)
```

```
throws OffsetOutOfBoundsException, SizeOut
OfBoundsException
```

Get the `short` at the given offset.

*Parameters:*

`offset` - The offset at which to read the short.

*Returns:* The short read.

*Throws:*

`SizeOutOfBoundsException`*217*,
`OffsetOutOfBoundsException`*217*

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

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

*Throws:*

`SizeOutOfBoundsException`*217*,
`OffsetOutOfBoundsException`*217*

```
public long map()
```

Maps the physical memory range into virtual memory. No-op if the system doesn't support virtual memory.

```
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.

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

Maps the physical memory range into virtual memory. No-op if the system doesn't support virtual memory.

*Parameters:*

`base` - The location to map at the virtual memory space.

`size` - Teh size of the block to map in.

```
public void setByte(long offset, byte value)
               throws OffsetOutOfBoundsException, SizeOut
               OfBoundsException
```

Set the `byte` at the given offset.

*Parameters:*

`offset` - The offset at which to write the byte.

`value` - The byte to write.

*Throws:*

`SizeOutOfBoundsException`$_{217}$, `OffsetOutOfBoundsException`$_{217}$

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

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

*Throws:*

`SizeOutOfBoundsException`$_{217}$, `OffsetOutOfBoundsException`$_{217}$

```
public void setInt(long offset, int value)
               throws OffsetOutOfBoundsException, SizeOut
               OfBoundsException
```

Set the `int` at the given offset.

*Parameters:*

`offset` - The offset at which to write the int.

`value` - The integer to write.

*Throws:*

`SizeOutOfBoundsException`$_{217}$, `OffsetOutOfBoundsException`$_{217}$

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

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

*Throws:*

`SizeOutOfBoundsException`[217], `OffsetOutOfBoundsException`[217]

```
public void setLong(long offset, long value)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Set the `long` at the given offset.

*Parameters:*

`offset` - The offset at which to write the long.

`value` - The long to write.

*Throws:*

`SizeOutOfBoundsException`[217], `OffsetOutOfBoundsException`[217]

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

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

*Throws:*

`SizeOutOfBoundsException`[217], `OffsetOutOfBoundsException`[217]

```
public void setShort(long offset, short value)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Set the `short` at the given offset.

*Parameters:*

`offset` - The offset at which to write the short.

`value` - The short to write.

*Throws:*

`SizeOutOfBoundsException`[217], `OffsetOutOfBoundsException`[217]

```
public void setShorts(long offset, short[] shorts,
               int low, int number)
               throws OffsetOutOfBoundsException, SizeOut
               OfBoundsException
```

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

*Throws:*

`SizeOutOfBoundsException`217, `OffsetOutOfBoundsException`217

```
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`117

*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`217 . 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`217 . 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, OffsetOutOfBound
               sException, SizeOutOfBoundsException, Unsu
               pportedPhysicalMemoryException, MemoryType
               ConflictException
```

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.

`OffsetOutOfBoundsException`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

`MemoryTypeConflictException`$_{215}$

```
public RawMemoryFloatAccess(java.lang.Object type,
            long base, long size)
            throws SecurityException, OffsetOutOfBound
            sException, SizeOutOfBoundsException, Unsu
            pportedPhysicalMemoryException, MemoryType
            ConflictException
```

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.

`OffsetOutOfBoundsException`$_{217}$ - The address is invalid.

`SizeOutOfBoundsException`$_{217}$ - The size is negative or extends into an invalid range of memory.

`UnsupportedPhysicalMemoryException`$_{218}$ - Thrown if the underlying hardware does not support the given type.

MemoryTypeConflictException$_{215}$

### 5.14.2 Methods

```
public double getDouble(long offset)
        throws OffsetOutOfBoundsException, SizeOut
        OfBoundsException
```

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.

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

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.

```
public float getFloat(long offset)
        throws OffsetOutOfBoundsException, SizeOut
        OfBoundsException
```

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.

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

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

```
public void setDouble(long offset, double value)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Set 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.

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

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.

```
public void setFloat(long offset, float value)
            throws OffsetOutOfBoundsException, SizeOut
            OfBoundsException
```

Set 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.

```
public void setFloats(long offset, float[] floats,
               int low, int number)
               throws OffsetOutOfBoundsException, SizeOut
               OfBoundsException
```

Set `number` `float` values starting at the given `offset` in this from the `byte` 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.

## 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

```
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`

```
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

```
public long getAllocationRate()
```

Get the allocation rate. Units are in bytes per second.

```
public long getMaxImmortal()
```

Get the limit on the amount of memory the thread may allocate in the immortal area. Units are in bytes.

```
public long getMaxMemoryArea()
```

Get the limit on the amount of memory the thread may allocate in the memory area. Units are in bytes.

```
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.

```
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.

```
public boolean setMaxImmortalIfFeasible(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.

`public boolean` **`setMaxMemoryAreaIfFeasible`**`(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* :

`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

`public` **`GarbageCollector`**`()`

### 5.16.2 Methods

public abstract `RelativeTime`$_{156}$ **`getPreemptionLatency`**()

Preemption latency is a measure of the maximum time a `RealtimeThread`$_{23}$ may have to wait for the collector to reach a preemption-safe point. Instances of `RealtimeThread`$_{23}$ are allowed to preeempt the garbage collector (instances of `NoHeapRealtimeThread`$_{33}$ preempt immediately but instances of `RealtimeThread`$_{23}$ 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

Chapter 6

# Synchronization



This section contains classes that:

- Allow the application of the priority ceiling emulation algorithm to individual objects.
- Allow the setting of the system default priority inversion algorithm.
- Allow wait-free communication between real-time threads and regular Java threads.

The specification strengthens the semantics of Java synchronization for use in real-time systems by mandating monitor execution eligibility control, commonly referred to as priority inversion control. A `MonitorControl` class is defined as the superclass of all such execution eligibility control algorithms. `PriorityInheritance` is the default monitor control policy; the specification also defines a `PriorityCeilingEmulation` option.

The wait-free queue classes provide protected, concurrent access to data shared between instances of `java.lang.Thread` and `NoHeapRealtimeThread`.

## Semantics and Requirements

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

1. Threads waiting to enter synchronized blocks are priority queue ordered. If threads with the same priority are possible under the active scheduling policy such threads are queued in FIFO order.
2. Any conforming implementation must provide an implementation of the 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* :

`public abstract class` **`MonitorControl`**

*Direct Known Subclasses:* `PriorityCeilingEmulation`~138~, `PriorityInheritance`~138~

*Description* :
Abstract superclass for all monitor control policy objects.

### 6.1.1 Constructors

```
public MonitorControl()
```

The default constructor.

### 6.1.2 Methods

```
public static MonitorControl136 getMonitorControl()
```

Return the system default monitor control policy.

```
public static MonitorControl136
        getMonitorControl(java.lang.Object
        monitor)
```

Return the monitor control policy for the given object.

```
public static void setMonitorControl(MonitorControl136
        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, MonitorControl136 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* :
`public class` **`PriorityCeilingEmulation`** `extends MonitorControl`[136]

*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`[136] and `PriorityInheritance`[138] .

### 6.2.1 Constructors

`public` **`PriorityCeilingEmulation`**`(int ceiling)`

Create a `PriorityCeilingEmulation` object with a given ceiling.

*Parameters:*

`ceiling` - Priority ceiling value.

### 6.2.2 Methods

`public int` **`getDefaultCeiling`**`()`

Get the priority ceiling for this `PriorityCeilingEmulation` object.

---

## 6.3 PriorityInheritance

*Declaration* :
`public class` **`PriorityInheritance`** `extends MonitorControl`[136]

*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`[136] and `PriorityCeilingEmulation`[138]

### 6.3.1 Constructors

```
public PriorityInheritance()
```

### 6.3.2 Methods

public static `PriorityInheritance`$_{138}$ **`instance`**`()`

Return a pointer to the singleton `PriorityInheritance`.

---

## 6.4 WaitFreeWriteQueue

*Declaration* :

```
public class WaitFreeWriteQueue
```

*Description* :

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread`$_{23}$ 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

public **`WaitFreeWriteQueue`**`(java.lang.Thread writer, java.lang.Thread reader, int maximum, MemoryArea`$_{77}$` 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`[77] 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`[216]

`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.

```
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.

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

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`$_{216}$

## 6.5 WaitFreeReadQueue

*Declaration* :

```
public class WaitFreeReadQueue
```

*Description* :

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread`$_{23}$ 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

```
public WaitFreeReadQueue(java.lang.Thread writer,
              java.lang.Thread reader, int maximum,
```

```
MemoryArea77 memory)
throws IllegalArgumentException, Instantia
tionException, ClassNotFoundException, Ill
egalAccessException
```

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`$_{77}$ in which this object and internal elements are stored.

*Throws:*

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

```
public WaitFreeReadQueue(java.lang.Thread writer,
          java.lang.Thread reader, int maximum,
          MemoryArea77 memory, boolean notify)
          throws IllegalArgumentException, Instantia
          tionException, ClassNotFoundException, Ill
          egalAccessException
```

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`$_{77}$ 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

`public void` **`clear`**`()`

Set this to empty.

`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`**`()`

Returns the next element in the queue unless the queue is empty. If the queue is empty null is returned.

`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.

`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`$_{183}$ which has `noHeap` characteristics.

`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`$_{216}$

## 6.6 WaitFreeDequeue

*Declaration* :

`public class` **`WaitFreeDequeue`**

*Description* :

The wait-free queue classes facilitate communication and synchronization between instances of `RealtimeThread`$_{23}$ and `java.lang.Thread`. See `WaitFreeWriteQueue`$_{139}$ or `WaitFreeReadQueue`$_{141}$ for more details. Instances of this class create a `WaitFreeWriteQueue`$_{139}$ and a `WaitFreeReadQueue`$_{141}$ and make calls on the respective `read()` and `write()` methods.

### 6.6.1 Constructors

```
public WaitFreeDequeue(java.lang.Thread writer,
        java.lang.Thread reader, int maximum,
        MemoryArea77 area)
        throws IllegalArgumentException, IllegalAc
        cessException, ClassNotFoundException, Ins
        tantiationException
```

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`$_{141}$ and the `WaitFreeWriteQueue`$_{139}$.

`area` - The `MemoryArea`$_{77}$ in which this object and internal elements are allocated.

*Throws:*

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

### 6.6.2 Methods

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

A synchronized call of the `read()` method of the underlying `WaitFreeWriteQueue`$_{139}$. 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.

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

A synchronized call of the `write()` method of the underlying `WaitFreeReadQueue`$_{141}$. 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`$_{216}$

`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.

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

An unsynchronized call of the `read()` method of the underlying `WaitFreeReadQueue`$_{141}$.

*Returns:* A `java.lang.Object` object read from this. If there are no elements in this then null is returned.

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

An unsynchronized call of the `write()` method of the underlying `WaitFreeWriteQueue`$_{139}$. 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`*216*

# Chapter 7
# Time



This section contains classes that:

- Allow description of a point in time with up to nanosecond accuracy and precision (actual accuracy and precision is dependent on the precision of the underlying system).
- Allow distinctions between absolute points in time, times relative to some starting point, and a new construct, rational time, which allows the efficient expression of occurrences per some interval of relative time.

The time classes required by the specification are `HighResolutionTime`, `AbsoluteTime, RelativeTime`, and `RationalTime.`

Instances of `HighResolutionTime` are not created, as the class exists to provide an implementation of the other three classes. An instance of `AbsoluteTime` encapsulates an absolute time expressed relative to midnight January 1, 1970 GMT. An instance of `RelativeTime` encapsulates a point in time that is relative to some other time value. Instances of `RationalTime` express a frequency by a numerator of type `long` (the frequency) and a denominator of type `RelativeTime.` If instances of `RationalTime` are given to certain constructors or methods the activity occurs for frequency times every interval. For example, if a `PeriodicTimer` is given an instance of `RationalTime` of (29,232) then the system will guarantee that the timer will fire exactly 29 times every 232 milliseconds even if the system has to slightly adjust the time between firings.

## Semantics and Requirements

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

1. All time objects must maintain nanosecond precision and report their values in terms of millisecond and nanosecond constituents.
2. Time objects must be constructed from other time objects, or from millisecond/nanosecond values.
3. Time objects must provide simple addition and subtraction operations, both for the entire object and for constituent parts.
4. Time objects must implement the `Comparable` interface if it is available. The `compareTo()` method must be implemented even if the interface is not available.
5. Any method of constructor that accepts a `RationalTime` of (x,y) must guarantee that its activity occurs exactly x times in every y milliseconds even if the intervals between occurrences of the activity have to be adjusted slightly. The RTSJ does not impose any required distribution on the lengths of the intervals but strongly suggests that implementations attempt to make them of approximately equal lengths.

## 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* :

```
public abstract class HighResolutionTime implements
```

java.lang.Comparable

*All Implemented Interfaces:* java.lang.Comparable

*Direct Known Subclasses:* `AbsoluteTime`$_{152}$, `RelativeTime`$_{156}$

*Description* :
Class `HighResolutionTime` is the base class for AbsoluteTime, RelativeTime, RationalTime.

### 7.1.1 Methods

`public abstract AbsoluteTime`$_{152}$ **`absolute`**`(Clock`$_{166}$ `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`$_{152}$ **`absolute`**`(Clock`$_{166}$ `clock, AbsoluteTime`$_{152}$ `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`$_{148}$ `time)`

Compares this HighResolutionTime with the specified HighResolutionTime.

*Parameters:*

`time` - compares with this time.

`public int` **`compareTo`**`(java.lang.Object object)`

For the Comparable interface.

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

`public boolean` **`equals`**`(HighResolutionTime`[148] `time)`

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

*Parameters:*

`time` - Values are compared to this.

`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.

`public final long` **`getMilliseconds`**`()`

Returns the milliseconds component of this.

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

`public final int` **`getNanoseconds`**`()`

Returns nanoseconds component of this.

`public int` **`hashCode`**`()`

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

`public abstract RelativeTime`[156] **`relative`**`(Clock`[166] `clock)`

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

`public abstract RelativeTime`[156] **`relative`**`(Clock`[166] `clock, HighResolutionTime`[148] `time)`

Convert the given instance of `HighResolutionTime` to an instance of `RelativeTime` relative to the given instance of `Clock`.

`public void` **`set`**`(HighResolutionTime`$_{148}$ `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.

`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.

`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.

`public static void` **`waitForObject`**`(java.lang.Object target,`
`HighResolutionTime`$_{148}$ `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 threads interrupts this thread while its waiting.

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

---

## 7.2 AbsoluteTime

*Declaration* :

`public class` **`AbsoluteTime`** `extends HighResolutionTime`$_{148}$

*All Implemented Interfaces:* java.lang.Comparable

*Description* :

An object that represents a specific point in time given by milliseconds plus nanoseconds past the epoch (January 1, 1970, 00:00:00 GMT). This representation was designed to be compatible with the standard Java representation of an absolute time in the `java.util.Date` class.

If the value of any of the millisecond or nanosecond fields is negative the variable is set to negative value. Although logically this may represent time before the epoch, invalid results may occur if an instance of `AbsoluteTime` representing time before the epoch is given as a parameter to the a method. For `add` and `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

`public` **`AbsoluteTime`**`()`

Equal to new AbsoluteTime(0,0).

`public` **`AbsoluteTime`**`(AbsoluteTime`$_{152}$ `time)`

Make a new `AbsoluteTime` object from the given `AbsoluteTime` object.

*Parameters:*

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

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

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

*Parameters:*

`date` - The java.util.Data representation of the time past the epoch

`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

`public AbsoluteTime`$_{152}$ **`absolute`**`(Clock`$_{166}$ `clock)`

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

*Overrides:* `public abstract AbsoluteTime`$_{152}$ `absolute(Clock`$_{166}$ `clock)`$_{149}$ in class `HighResolutionTime`$_{148}$

*Parameters:*

`clock` - Clock on which this is based

*Returns:* this

`public AbsoluteTime`$_{152}$ **`absolute`**`(Clock`$_{166}$ `clock, AbsoluteTime`$_{152}$ `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.

*Overrides:* `public abstract AbsoluteTime`152 `absolute(Clock`166 `clock, AbsoluteTime`152 `dest)`*149* in class `HighResolutionTime`*148*

*Parameters:*

`clock` - Clock on which this is based

`destination` - Converted to an absolute time

*Returns:* this

`public AbsoluteTime`*152* **`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`*152* **`add`**`(long millis, int nanos, AbsoluteTime`*152* `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`*152* **`add`**`(RelativeTime`*156* `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`*152* **`add`**`(RelativeTime`*156* `time, AbsoluteTime`*152* `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$_{156}$ **relative**(Clock$_{166}$ clock)

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

*Overrides:* public abstract RelativeTime$_{156}$ relative(Clock$_{166}$ clock)$_{150}$ in class HighResolutionTime$_{148}$

public RelativeTime$_{156}$ **relative**(Clock$_{166}$ clock, AbsoluteTime$_{152}$ destination)

Convert the given instance of `RelativeTime` to an instance of `RelativeTime` 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$_{156}$ **subtract**(AbsoluteTime$_{152}$ time)

*Parameters:*

`time` - absolute time to subtract from `this`

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

public final RelativeTime$_{156}$ **subtract**(AbsoluteTime$_{152}$ time, RelativeTime$_{156}$ destination)

*Parameters:*

`time` - absolute 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 final AbsoluteTime`$_{152}$ **`subtract`**`(RelativeTime`$_{156}$ `time)`

*Parameters:*

`time` - relative time to subtract from `this`

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

`public AbsoluteTime`$_{152}$ **`subtract`**`(RelativeTime`$_{156}$ `time,`
`AbsoluteTime`$_{152}$ `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, in a format that matches java.util.Date.toString() with a postfix to the detail the sub-second value

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

*Returns:* String object converted from this.

## 7.3 RelativeTime

*Declaration* :
`public class` **`RelativeTime`** `extends HighResolutionTime`$_{148}$

*All Implemented Interfaces:* java.lang.Comparable

*Direct Known Subclasses:* `RationalTime`$_{160}$

*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

public **RelativeTime**()

Equivalent to new RelativeTime(0,0)

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

public **RelativeTime**(RelativeTime$_{156}$ 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

public AbsoluteTime$_{152}$ **absolute**(Clock$_{166}$ clock)

*Overrides:* public abstract AbsoluteTime$_{152}$ absolute(Clock$_{166}$ clock)$_{149}$ in class HighResolutionTime$_{148}$

public AbsoluteTime$_{152}$ **absolute**(Clock$_{166}$ clock, AbsoluteTime$_{152}$ 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$_{152}$ absolute(Clock$_{166}$ clock, AbsoluteTime$_{152}$ dest)$_{149}$ in class HighResolutionTime$_{148}$

*Parameters:*

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

`public RelativeTime`[156] **`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`[156] **`add`**`(long millis, int nanos,`
`RelativeTime`[156] `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`[156] **`add`**`(RelativeTime`[156] `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`[156] **`add`**`(RelativeTime`[156] `time,`
`RelativeTime`[156] `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`$_{152}$ `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`$_{156}$ **`getInterarrivalTime`**`()`

Return the interarrival time that is the result of dividing this interval by its frequency. For a `RelativeTime`, and `RationalTime`$_{160}$ s with a frequency of 1, it just returns this. The interarrival time is necessarily an approximation.

`public RelativeTime`$_{156}$ **`getInterarrivalTime`**`(RelativeTime`$_{156}$ `destination)`

Return the interarrival time that is the result of dividing this interval by its frequency. For a `RelativeTime`, or a `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`$_{156}$ **`relative`**`(Clock`$_{166}$ `clock)`

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

*Overrides:* `public abstract RelativeTime`$_{156}$ `relative(Clock`$_{166}$ `clock)`$_{150}$ in class `HighResolutionTime`$_{148}$

`public RelativeTime`$_{156}$ **`relative`**`(Clock`$_{166}$ `clock, RelativeTime`$_{156}$ `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`$_{156}$ **`subtract`**`(RelativeTime`$_{156}$ `time)`

*Parameters:*

`time` - relative time to subtract from `this`

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

```
public RelativeTime156 subtract(RelativeTime156 time,
                  RelativeTime156 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 RelativeTime156`

*All Implemented Interfaces:* java.lang.Comparable

*Description* :

An object that represents a time interval 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

```
public RationalTime(int frequency)
```

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

```
public RationalTime(int frequency, long millis,
            int nanos)
            throws IllegalArgumentException
```

Construct a new Object of RationalTime. All arguments must be >= 0.

*Parameters:*

`frequency` - The frequency value of this

`millis` - The milliseonds value of this

`nanos` - The nanoseconds value of this

*Throws:*

`IllegalArgumentException`

```
public RationalTime(int frequency,
            RelativeTime156 interval)
            throws IllegalArgumentException
```

Construct a new Object of RationalTime from the given RelativeTime

*Parameters:*

`frequency` - The frequency value of this

`interval` - The relativeTime object used as the source for the copy

*Throws:*

`IllegalArgumentException`

### 7.4.2 Methods

```
public AbsoluteTime152 absolute(Clock166 clock,
            AbsoluteTime152 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 AbsoluteTime`$_{152}$ `absolute(Clock`$_{166}$ `clock, AbsoluteTime`$_{152}$ `destination)`$_{157}$ in class `RelativeTime`$_{156}$

*Parameters:*

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

`public void` **`addInterarrivalTo`**`(AbsoluteTime`$_{152}$ `destination)`

Add this time to an `AbsoluteTime`$_{152}$ . It is almost the same dest.add(this, dest) except that it accounts for(ie. divides by) the frequency.

*Overrides:* `public void addInterarrivalTo(AbsoluteTime`$_{152}$ `destination)`$_{159}$ in class `RelativeTime`$_{156}$

*Parameters:*

`public int` **`getFrequency`**`()`

Return the frquency of this.

`public RelativeTime`$_{156}$ **`getInterarrivalTime`**`()`

Gets the time duration between two consecutive ticks using frequency

*Overrides:* `public RelativeTime`$_{156}$ `getInterarrivalTime()`$_{159}$ in class `RelativeTime`$_{156}$

`public RelativeTime`$_{156}$ **`getInterarrivalTime`**`(RelativeTime`$_{156}$ `dest)`

Gets the time duration between two consecutive ticks using frequency

*Overrides:* `public RelativeTime`$_{156}$ `getInterarrivalTime(RelativeTime`$_{156}$ `destination)`$_{159}$ in class `RelativeTime`$_{156}$

*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

*Overrides:* `public void set(long millis, int nanos)`$_{151}$ in class `HighResolutionTime`$_{148}$

*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 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`$_{166}$ **`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`$_{156}$ **`getResolution`**`()`

Return the resolution of the clock —- the interval between ticks.

*Returns:* A RelativeTime object representing the resolution of this

`public AbsoluteTime`$_{152}$ **`getTime`**`()`

Return the current time in a freshly allocated object.

*Returns:* An AbsoluteTime object representing the current time.

`public abstract void` **`getTime`**`(AbsoluteTime`$_{152}$ `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.

```
public abstract void setResolution(RelativeTime156
        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* :

```
public abstract class Timer extends AsyncEvent181
```

*Direct Known Subclasses:* `OneShotTimer`$_{170}$, `PeriodicTimer`$_{171}$

*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

```
protected Timer(HighResolutionTime148 t, Clock166 c,
        AsyncEventHandler183 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.

`handler` - The default handler to use for this event. If null, no handler is associated with it and nothing will happen when this event fires until a handler is provided

### 8.2.2 Methods

`public ReleaseParameters`$_{54}$ **`createReleaseParameters`**`()`

Create a `ReleaseParameters`$_{54}$ block appropriate to the timing characteristics of this event. The default is the most pessimistic: `AperiodicParameters`$_{59}$ . 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 ReleaseParameters`$_{54}$ `createReleaseParameters()`$_{182}$ in class `AsyncEvent`$_{181}$

*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 reabled before its fire time occures 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()`

`public Clock166 getClock()`

Return the Clock that this timer is based on

*Returns:* clock The clock of this timer based on

`public AbsoluteTime152 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(HighResolutionTime148 time)`

Change the scheduled time for this event. can take either absolute or relative 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 Timer168`

*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(HighResolutionTime148 time,
            AsyncEventHandler183 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(HighResolutionTime148 start,
            Clock166 clock, AsyncEventHandler183 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 Timer168`

*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

public **PeriodicTimer**(HighResolutionTime$_{148}$ start,
RelativeTime$_{156}$ interval,
AsyncEventHandler$_{183}$ handler)

Create an instance of AsyncEvent that executes its fire method periodiacally

*Parameters:*

`start` - The time when the first interval begins

`interval` - The time between successive executions of the fire method

`handler` - The instance of AsyncEventHandler that will be scheduled each time the fire method is exceuted

public **PeriodicTimer**(HighResolutionTime$_{148}$ start,
RelativeTime$_{156}$ interval, Clock$_{166}$ clock,
AsyncEventHandler$_{183}$ 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

public ReleaseParameters$_{54}$ **createReleaseParameters**()

Create a `ReleaseParameters`$_{54}$ object with the next fire time as the start time and the interval of `this` as the period.

*Overrides:* `public ReleaseParameters`$_{54}$ `createReleaseParameters()`$_{169}$ in class `Timer`$_{168}$

*Returns:* an instance of ReleaseParameters object

`public void` **`fire`**`()`

Causes the instance of the superclass `AsyncEvent`$_{181}$ to occur now.

*Overrides:* `public void fire()`$_{182}$ in class `AsyncEvent`$_{181}$

`public AbsoluteTime`$_{152}$ **`getFireTime`**`()`

Return the next time at which this will fire.

*Overrides:* `public AbsoluteTime`$_{152}$ `getFireTime()`$_{170}$ in class `Timer`$_{168}$

`public RelativeTime`$_{156}$ **`getInterval`**`()`

Return the interval of this Timer

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

`public void` **`setInterval`**`(RelativeTime`$_{156}$ `interval)`

Reset the interval of this Timer

*Parameters:*

`interval` - A RelativeTime object which is the interval to reset this Timer

Chapter 9

# Asynchrony



This section contains classes that:

- Provide mechanisms that bind the execution of program logic to the occurrence of internal and external events.
- Provide mechanisms that allow the asynchronous transfer of control.
- Provide mechanisms that allow the asynchronous termination of threads.

This specification provides several facilities for arranging asynchronous control of execution, some of which apply to threads in general while others apply only to real-time threads. These facilities fall into two main categories: asynchronous event handling and asynchronous transfer of control (ATC), which includes thread termination.

Asynchronous event handling is captured by the non-abstract class `AsyncEvent` and the abstract classes `AsyncEventHandler` and `BoundAsyncEventHandler`. An instance of the `AsyncEvent` class is an object corresponding to the possibility of an asynchronous event occurrence. An event occurrence may be initiated by either application logic or by the occurrence of a *happening* external to the JVM (such as a software signal or a hardware interrupt handler). An event occurrence is expressed in program logic by the invocation of the `fire()` method of an instance of the `AsyncEvent` class. The initiation of an event occurrence due to a happening is implementation dependent.

An instance of the class `AsyncEventHandler` is an object embodying code that is scheduled in response to the occurrence of an event. The `run()` method of an instance of `AsyncEventHandler` acts like a thread, and indeed one of its constructors takes references to instances of `SchedulingParameters`, `ReleaseParameters`, and `MemoryParameters`. However, there is not necessarily a separate thread for each `run()` method. The class `BoundAsyncEventHandler` extends `AsyncEventHandler`, and should be used if it is necessary to ensure that a handler has a dedicated thread. An event count is maintained so that a handler can cope with event bursts —- situations where an event is fired more frequently than its handler can respond.

The `interrupt()` method in `java.lang.Thread` provides rudimentary asynchronous communication by setting a pollable/resettable flag in the target thread, and by throwing a synchronous exception when the target thread is blocked at an invocation of `wait()`, `sleep()`, or `join()`. This specification extends the effect of `Thread.interrupt()` and adds an overloaded version in `RealtimeThread`, offering a more comprehensive and non-polling asynchronous execution control facility. It is based on throwing and propagating exceptions that, though asynchronous, are deferred where necessary in order to avoid data structure corruption. The main elements of ATC are embodied in the class `AsynchronouslyInterruptedException` (AIE), its subclass `Timed`, the interface `Interruptible`, and in the semantics of the interrupt methods in `Thread` and `RealtimeThread`.

A method indicates its willingness to be asynchronously interrupted by including AIE on its `throws` clause. If a thread is asynchronously interrupted while executing a method that identifies AIE on its `throws` clause, then an instance of AIE will be thrown as soon as the thread is outside of a section in which ATC is deferred. Several idioms are available for handling an AIE, giving the programmer the choice of using `catch` clauses and a low-level mechanism with specific control over propagation, or a higher-level facility that allows specifying the interruptible code, the handler, and the result retrieval as separate methods.

## Semantics and Requirements

This list establishes the semantics and requirements that are applicable to `AsyncEvent` objects. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. When an instance of `AsyncEvent` occurs (by either program logic or a happening), all `run()` methods of instances of the `AsyncEventHandler` class that have been added to the instance of `AsyncEvent` by the execution of `addHandler()` are scheduled for execution. This action may or may not be idempotent. Every occurrence of an event increments a counter in each associated handler. Handlers may elect to execute logic for each occurrence of the event or not.

2. Instances of `AsyncEvent` and `AsyncEventHandler` may be created and used by any program logic.

3. More than one instance of `AsyncEventHandler` may be added to an instance of `AsyncEvent`.

4. An instance of `AsyncEventHandler` may be added to more than one instance of `AsyncEvent`.

This list establishes the semantics and requirements that are applicable to `AsynchronouslyInterruptedException`. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. Instances of the class `AsynchronouslyInterruptedException` can be generated by execution of program logic and by internal virtual machine mechanisms that are asynchronous to the execution of program logic which is the target of the exception.

2. Program logic that exists in methods that throw `AsynchronouslyInterruptedException` is subject to receiving an instance of `AsynchronouslyInterruptedException` at any time during execution except as provided below.

3. The RTSJ specifically requires that blocking methods in `java.io.*` must be prevented from blocking indefinitely when invoked from a method with AIE in its `throws` clause. The implementation, when either `AIE.fire()` or `RealtimeThread.interrupt()` is called when control is in a `java.io.*` method invoked from an interruptible method, may either unblock the blocked call, raise an `IOException` on behalf of the call, or allow the call to complete normally if the implementation determines that the call would eventually unblock.

4. 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:

1. Control enters an AI-method.
2. Control returns to an AI-method.
3. Control leaves a synchronized block within an AI-method.

When `happened()` is called on an AIE or that AIE is superseded by another the first AIE's state is made non-pending.

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

| | Match | No Match |
|---|---|---|
| Propagate == true | clear the pending AIE, return true | propagate (whether the AIE remains pending is invisible except to the implementation) |
| Propagate == false | clear the pending AIE, return false | do not clear the pending AIE, return false |

precedence among active instances of AIE. Let $AIE_0$ be the AIE raised when `t.interrupt()` is invoked and $AIE_i$ (i = 1,...,n, for n unique instances of AIE) be the AIE raised when $AIE_i$`.fire()` is invoked. Assume stacks grow down and therefore the phrase “a frame lower on the stack than this frame” refers to a method at a deeper nesting level.

1. If the current AIE is an $AIE_0$ and the new AIE is an $AIE_x$ associated with any frame on the stack then the new AIE ($AIE_x$) is discarded.
2. If the current AIE is an $AIE_x$ and the new AIE is an $AIE_0$, then the current AIE ($AIE_x$) is replaced by the new AIE ($AIE_0$).
3. If the current AIE is an $AIE_x$ and the new AIE is an $AIE_y$ from a frame lower on the stack, then the new AIE ($AIE_y$) discarded.
4. If the current AIE is an $AIE_x$ and the new AIE is an $AIE_y$ from a frame higher on the stack, the current AIE ($AIE_x$) is replaced by the new AIE ($AIE_y$).

### Rationale

The design of the asynchronous event handling was intended to provide the necessary functionality while allowing efficient implementations and catering to a variety of real-time applications. In particular, in some real-time systems there may be a large number of potential events and event handlers (numbering in the thousands or perhaps even the tens of thousands), although at any given time only a small number will be used. Thus it would not be appropriate to dedicate a thread to each event handler. The RTSJ addresses this issue by allowing the programmer to specify an event handler either as not bound to a specific thread (the class `AsyncEventHandler`) or alternatively as bound to a thread (`BoundAsyncEventHandler`).

Events are dataless: the fire method does not pass any data to the handler. This was intentional in the interest of simplicity and efficiency. An application that needs to associate data with an `AsyncEvent` can do so explicitly by setting up a buffer; it will then need to deal with buffer overflow issues as required by the application.

The ability for one thread to trigger an ATC in another thread is necessary in many kinds of real-time applications but must be designed carefully in order to minimize the risks of problems such as data structure corruption and deadlock. There is, invariably, a tension between the desire to cause an ATC to be immediate, and the desire to ensure that certain sections of code are executed to completion.

One basic decision was to allow ATC in a method only if the method explicitly permits this. The default of no ATC is reasonable, since legacy code might be written expecting no ATC, and asynchronously aborting the execution of such a method could lead to unpredictable results. Since the natural way to model ATC is with an exception (`AsynchronouslyInterruptedException`, or AIE), the way that a method indicates its susceptibility to ATC is by including AIE on its `throws` clause. Causing this exception to be thrown in a thread `t` as an effect of calling `t.interrupt()` was a natural extension of the semantics of interrupt as currently defined by `java.lang.Thread`.

One ATC-deferred section is `synchronized` code. This is a context that needs to be executed completely in order to ensure a program operates correctly. If `synchronized` code were aborted, a shared object could be left in an inconsistent state.

Constructors and `finally` clauses are subject to interruption. If a constructor is aborted, an object might be only partially initialized. If a `finally` clause is aborted, needed cleanup code might not be performed. It is the programmer's responsibility to ensure that executing these constructs does not induce unwanted ATC latency. Note that by making synchronized code ATC-deferred, this specification avoids the problems that caused `Thread.stop()` to be deprecated and that have made the use of `Thread.destroy()` prone to deadlock.

A potential problem with using the exception mechanism to model ATC is that a method with a "catch-all" handler (for example a `catch` clause identifying `Exception` or even `Throwable` as the exception class) can inadvertently intercept an exception intended for a caller. This problem is avoided by having special semantics for catching an instance of AIE. Even though a catch clause may catch an AIE, the exception will be propagated unless the handler invokes the happened method from AIE. Thus, if a thread is asynchronously interrupted while in a try block that has a handler such as

```
catch (Throwable e){ return; }
```

then the AIE instance will still be propagated to the caller.

This specification does not provide a special mechanism for terminating a thread; ATC can be used to achieve this effect. This means that, by default, a thread cannot be terminated; it needs to invoke methods that have AIE in their `throws` clauses. Allowing termination as the default would have been questionable, bringing the same insecurities that are found in `Thread.stop()` and `Thread.destroy()`.

## 9.1 AsyncEvent

*Declaration* :

`public class` **`AsyncEvent`**

*Direct Known Subclasses:* `Timer`$_{168}$

*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`$_{183}$ and `Scheduler`$_{45}$ ).

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`$_{183}$ `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 `happening` are implementation dependent. This `AsyncEvent` is considered to have occurred whenever the external event occurs.

*Parameters:*

`happening` - An implementation dependent value that binds this `AsyncEvent` to some external event.

*Throws:*

`UnknownHappeningException`$_{220}$ - if the happening string is not supported by the system.

### `public ReleaseParameters`$_{54}$ `createReleaseParameters()`

Create a `ReleaseParameters`$_{54}$ block appropriate to the timing characteristics of this event. The default is the most pessimistic: `AperiodicParameters`$_{59}$. 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.

### `public void fire()`

Fire (schedule the `run()` methods of) the handlers associated with this event.

### `public boolean handledBy(AsyncEventHandler`$_{183}$ `handler)`

Returns true if and only if this event is handled by this handler.

*Parameters:*

`target` - The handler to be tested to determine if it is associated with this. Returns false if `target` is null.

### `public void removeHandler(AsyncEventHandler`$_{183}$ `handler)`

Remove a handler from the set associated with this event.

*Parameters:*

`handler` - The handler to be disassociated from this. If null nothing happens. If not already associated with this then nothing happens.

### `public void setHandler(AsyncEventHandler`$_{183}$ `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).

```
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`$_{220}$ - 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* :

`public class` **`AsyncEventHandler`** `implements Schedulable`$_{41}$

*All Implemented Interfaces:* java.lang.Runnable, `Schedulable`$_{41}$

*Direct Known Subclasses:* `BoundAsyncEventHandler`$_{195}$

*Description* :

An asynchronous event handler encapsulates code that gets run at some time after an `AsyncEvent`$_{181}$ occurs.

It is essentially a `java.lang.Runnable` with a set of parameter objects, making it very much like a `RealtimeThread`$_{23}$ . The expectation is that there may be thousands of events, with corresponding handlers, averaging about one handler per event. The number of unblocked (i.e., scheduled) handlers is expected to be relatively small.

It is guaranteed that multiple firings of an event handler will be serialized. It is also guaranteed that (unless the handler explicitly chooses otherwise) for each firing of the handler, there will be one execution of the `handleAsyncEvent()` method.

For instances of `AsyncEventHandler` with a release parameter of type `SporadicParameters`$_{61}$ have a list of release times which correspond to execution times of `AsyncEvent.fire()`. The minimum interarrival time specified in `SporadicParameters`$_{61}$ 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`$_{181}$ 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`$_{195}$.

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`$_{51}$ are inherited from the current thread and does not have either `ReleaseParameters`$_{54}$ or `MemoryParameters`$_{129}$.

### `public` **`AsyncEventHandler`**`(boolean nonheap)`

Create a handler whose parameters are inherited from the current thread, if it is a `RealtimeThread`$_{23}$, or null otherwise.

*Parameters:*

`nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`$_{33}$. A false value means this will have characteristics identical to a `RealtimeThread`$_{23}$. If true and the current thread is *not* a `NoHeapRealtimeThread`$_{33}$ or a `RealtimeThread`$_{23}$ executing within a `ScopedMemory`$_{84}$ or `ImmortalMemory`$_{82}$ 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$_{23}$ , or null otherwise.

*Parameters:*

nonheap - A flag meaning, when true, that this will have characteristics identical to a NoHeapRealtimeThread$_{33}$ . A false value means this will have characteristics identical to a RealtimeThread$_{23}$ . If true and the current thread is *not* a NoHeapRealtimeThread$_{33}$ or a RealtimeThread$_{23}$ executing within a ScopedMemory$_{84}$ or ImmortalMemory$_{82}$ 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$_{51}$ are inherited from the current thread and does not have either ReleaseParameters$_{54}$ or MemoryParameters$_{129}$ .

*Parameters:*

logic - The java.lang.Runnable object whose run is executed by handleAsyncEvent.

public **AsyncEventHandler**(SchedulingParameters$_{51}$ scheduling,
ReleaseParameters$_{54}$ release,
MemoryParameters$_{129}$ memory,
MemoryArea$_{77}$ area,
ProcessingGroupParameters$_{67}$ group,
boolean nonheap)

Create a handler with the specified parameters.

*Parameters:*

scheduling - A SchedulingParameters$_{51}$ object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters$_{51}$ of the current thread.

release - A ReleaseParameters$_{54}$ object which will be associated with the constructed instance of this. If null this will have no ReleaseParameters$_{54}$ .

`memory` - A `MemoryParameters`$_{129}$ object which will be associated with the constructed instance of this. If null this will have no `MemoryParameters`$_{129}$.

`area` - The `MemoryArea`$_{77}$ 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`$_{84}$ or `ImmortalMemory`$_{82}$ object if `noheap` is true.

`group` - A `ProcessingGroupParameters`$_{67}$ 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`$_{33}$.

`logic` - The `java.lang.Runnable` object whose `run` is executed by handleAsyncEvent.

*Throws:*

`{@link` - IllegalArgumentException} if the initial memory area is in heap memory, and the `noheap` parameter is true.

```
public AsyncEventHandler(SchedulingParameters51 scheduling,
            ReleaseParameters54 release,
            MemoryParameters129 memory,
            MemoryArea77 area,
            ProcessingGroupParameters67 group,
            boolean nonheap, java.lang.Runnable logic)
```

Create a handler with the specified parameters.

*Parameters:*

`scheduling` - A `SchedulingParameters`$_{51}$ object which will be associated with the constructed instance of this. If null this will be assigned the reference to the `SchedulingParameters`$_{51}$ of the current thread.

`release` - A `ReleaseParameters`$_{54}$ object which will be associated with the constructed instance of this. If null this will have no `ReleaseParameters`$_{54}$.

`memory` - A `MemoryParameters`$_{129}$ object which will be associated with the constructed instance of this. If null this will have no `MemoryParameters`$_{129}$.

`area` - The `MemoryArea`$_{77}$ 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`$_{84}$ or `ImmortalMemory`$_{82}$ object if `noheap` is true.

`group` - A `ProcessingGroupParameters`$_{67}$ 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`$_{33}$.

*Throws:*

`{@link` - IllegalArgumentException} if the initial memory area is in heap memory, and the `noheap` parameter is true.

```
public AsyncEventHandler(SchedulingParameters51 scheduling,
            ReleaseParameters54 release,
            MemoryParameters129 memory,
            MemoryArea77 area,
            ProcessingGroupParameters67 group,
            java.lang.Runnable logic)
```

Create a handler with the specified parameters.

*Parameters:*

`release` - A `ReleaseParameters`$_{54}$ object which will be associated with the constructed instance of this. If null this will have no `ReleaseParameters`$_{54}$.

`scheduling` - A `SchedulingParameters`$_{51}$ object which will be associated with the constructed instance of this. If null this will be assigned the reference to the `SchedulingParameters`$_{51}$ of the current thread.

`memory` - A `MemoryParameters`$_{129}$ object which will be associated with the constructed instance of this. If null this will have no `MemoryParameters`$_{129}$.

`area` - The `MemoryArea`$_{77}$ for this. If null the memory area will be that of the current thread.

`group` - A `ProcessingGroupParameters`$_{67}$ 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`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$) of this instance of `Schedulable`$_{41}$ will be considered in the feasibility analysis of the associated `Scheduler`$_{45}$ until further notice. Whether the resulting system is feasible or not, the addition is completed.

*Specified By:* `public boolean addToFeasibility()`$_{41}$ in interface `Schedulable`$_{41}$

*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:

```
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$_{77}$ **getMemoryArea**()

Get the current memory area.

*Returns:* The current memory area in which allocations occur.

### public MemoryParameters$_{129}$ **getMemoryParameters**()

Get the memory parameters associated with this handler.

*Specified By:* `public MemoryParameters`$_{129}$ `getMemoryParameters()`$_{42}$ in interface `Schedulable`$_{41}$

*Returns:* The `MemoryParameters`$_{129}$ object associated with this.

### protected final int **getPendingFireCount**()

Return the number of pending executions of this handler

*Returns:* The pending fire count.

### public ProcessingGroupParameters$_{67}$ **getProcessingGroupParameters**()

Returns a reference to the `ProcessingGroupParameters`$_{67}$ object.

*Specified By:* `public ProcessingGroupParameters`$_{67}$ `getProcessingGroupParameters()`$_{42}$ in interface `Schedulable`$_{41}$

### `public ReleaseParameters`$_{54}$ **`getReleaseParameters`**`()`

Get the release parameters associated with this handler.

*Specified By:* `public ReleaseParameters`$_{54}$ `getReleaseParameters()`$_{42}$ in interface `Schedulable`$_{41}$

*Returns:* The `ReleaseParameters`$_{54}$ object associated with this.

### `public Scheduler`$_{45}$ **`getScheduler`**`()`

Return the `Scheduler`$_{45}$ for this handler.

*Specified By:* `public Scheduler`$_{45}$ `getScheduler()`$_{42}$ in interface `Schedulable`$_{41}$

*Returns:* The instance of the scheduler managing this.

### `public SchedulingParameters`$_{51}$ **`getSchedulingParameters`**`()`

Returns a reference to the scheduling parameters object.

*Specified By:* `public SchedulingParameters`$_{51}$ `getSchedulingParameters()`$_{42}$ in interface `Schedulable`$_{41}$

*Returns:* The `SchedulingParameters`$_{51}$ 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`$_{51}$, `ReleaseParameters`$_{54}$, `MemoryParameters`$_{129}$, and `ProcessingGroupParameters`$_{67}$, of this instance of `Schedulable`$_{41}$ should no longer be considered in the feasibility analysis of the associated `Scheduler`$_{45}$. Whether the resulting system is feasible or not, the subtraction is completed.

*Specified By:* `public boolean removeFromFeasibility()`$_{42}$ in interface `Schedulable`$_{41}$

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

`public final void` **`run`**`()`

Used by the asynchronous event mechanism, see `AsyncEvent`$_{181}$ . 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.lang.Runnable

`public boolean` **`setIfFeasible`**`(ReleaseParameters`$_{54}$ `release, MemoryParameters`$_{129}$ `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`$_{54}$ `release, MemoryParameters`$_{129}$ `memory, ProcessingGroupParameters`$_{67}$ `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`$_{54}$ `release, ProcessingGroupParameters`$_{67}$ `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`$_{129}$ `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`$_{129}$ `memory)`$_{42}$ in interface `Schedulable`$_{41}$

*Parameters:*

`memory` - A `MemoryParameters`$_{129}$ object which will become the `MemoryParameters`$_{129}$ associated with this after the method call.

`public boolean` **`setMemoryParametersIfFeasible`**`(MemoryParameters`$_{129}$ `memory)`

*Specified By:* `public boolean setMemoryParametersIfFeasible(MemoryParameters`$_{129}$ `memParam)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setProcessingGroupParameters`**`(ProcessingGroupParameters`$_{67}$ `group)`

Sets the reference to the `ProcessingGroupParameters`$_{67}$ object.

*Specified By:* `public void setProcessingGroupParameters(ProcessingGroupParameters`$_{67}$ `groupParameters)`$_{43}$ in interface `Schedulable`$_{41}$

`public boolean` **`setProcessingGroupParametersIfFeasible`**`(ProcessingGroupParameters`$_{67}$ `group)`

*Specified By:* `public boolean setProcessingGroupParametersIfFeasible(ProcessingGroupParameters`$_{67}$ `groupParameters)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setReleaseParameters`**`(ReleaseParameters`$_{54}$ `release)`

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.

*Specified By:* `public void setReleaseParameters(ReleaseParameters`$_{54}$ `release)`$_{43}$ in interface `Schedulable`$_{41}$

*Parameters:*

`parameters` - A `ReleaseParameters`$_{54}$ object which will become the `ReleaseParameters`$_{54}$ associated with this after the method call.

`public boolean` **`setReleaseParametersIfFeasible`**`(ReleaseParameters`$_{54}$ `release)`

*Specified By:* `public boolean setReleaseParametersIfFeasible(ReleaseParameters`$_{54}$ `release)`$_{43}$ in interface `Schedulable`$_{41}$

`public void` **`setScheduler`**`(Scheduler`$_{45}$ `scheduler) throws IllegalThreadStateException`

Set the scheduler for this handler. A reference to the scheduler which will manage the execution of this thread.

*Specified By:* `public void setScheduler(Scheduler`$_{45}$ `scheduler) throws IllegalThreadStateException`$_{44}$ in interface `Schedulable`$_{41}$

*Parameters:*

`scheduler` - An instance of `Scheduler`$_{45}$ (or subclasses) which will manage the execution of this thread. If `scheduler` is null nothing happens.

*Throws:*

`IllegalThreadStateException`

`public void` **`setScheduler`**`(Scheduler`$_{45}$ `scheduler,`

SchedulingParameters$_{51}$ scheduling,
ReleaseParameters$_{54}$ release,
MemoryParameters$_{129}$ memoryParameters,
ProcessingGroupParameters$_{67}$ processingGroup)
throws IllegalThreadStateException

Set the scheduler for this handler. A reference to the scheduler which will manage the execution of this thread.

*Specified By:* public void setScheduler(Scheduler$_{45}$ scheduler, SchedulingParameters$_{51}$ scheduling, ReleaseParameters$_{54}$ release, MemoryParameters$_{129}$ memoryParameters, ProcessingGroupParameters$_{67}$ processingGroup) throws IllegalThreadStateException$_{44}$ in interface Schedulable$_{41}$

*Parameters:*

scheduler - An instance of Scheduler$_{45}$ (or subclasses) which will manage the execution of this thread. If scheduler is null nothing happens.

scheduling - A SchedulingParameters$_{51}$ object which will be associated with the constructed instance of this. If null this will be assigned the reference to the SchedulingParameters$_{51}$ of the current thread.

release - A ReleaseParameters$_{54}$ object which will be associated with the constructed instance of this. If null this will have no ReleaseParameters$_{54}$.

memory - A MemoryParameters$_{129}$ object which will be associated with the constructed instance of this. If null this will have no MemoryParameters$_{129}$.

group - A ProcessingGroupParameters$_{67}$ object to which this will be associated. If null this will not be associated with any processing group.

*Throws:*

IllegalThreadStateException

public void **setSchedulingParameters**(SchedulingParameters$_{51}$ scheduling)

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 setSchedulingParameters(SchedulingParameters`$_{51}$ `scheduling)`$_{44}$ in interface `Schedulable`$_{41}$

*Parameters:*

`parameters` - A `SchedulingParameters`$_{51}$ object which will become the `SchedulingParameters`$_{51}$ object associated with this after the method call.

`public boolean` **`setSchedulingParametersIfFeasible`**`(SchedulingParameters`$_{51}$ `sched)`

Set the `SchedulingParameters`$_{51}$ of this scheduable object only if the resulting task set is feasible.

*Specified By:* `public boolean setSchedulingParametersIfFeasible(SchedulingParameters`$_{51}$ `scheduling)`$_{44}$ in interface `Schedulable`$_{41}$

*Parameters:*

`scheduling` - The `SchedulingParameters`$_{51}$ object. If null nothing happens.

## 9.3 BoundAsyncEventHandler

*Declaration* :

`public abstract class` **`BoundAsyncEventHandler`** `extends AsyncEventHandler`$_{183}$

*All Implemented Interfaces:* java.lang.Runnable, `Schedulable`$_{41}$

*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`$_{23}$ , or null otherwise.

`public` **`BoundAsyncEventHandler`**`(SchedulingParameters`$_{51}$
`scheduling, ReleaseParameters`$_{54}$ `release,`
`MemoryParameters`$_{129}$ `memory,`
`MemoryArea`$_{77}$ `area,`
`ProcessingGroupParameters`$_{67}$ `group,`
`boolean nonheap, java.lang.Runnable logic)`

Create a handler with the specified `ReleaseParameters`$_{54}$ and `MemoryParameters`$_{129}$ .

*Parameters:*

- `scheduling` - A `SchedulingParameters`$_{51}$ object which will be associated with the constructed instance of this. If null this will be assigned the reference to the `SchedulingParameters`$_{51}$ of the current thread.
- `release` - The `ReleaseParameters`$_{54}$ object for this. A value of null will construct this without a `ReleaseParameters`$_{54}$ object.
- `memory` - The `MemoryParameters`$_{129}$ object for this. A value of null will construct this without a `MemoryParameters`$_{129}$ object.
- `area` - The `MemoryArea`$_{77}$ 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`$_{84}$ or `ImmortalMemory`$_{82}$ object if `noheap` is true.
- `nonheap` - A flag meaning, when true, that this will have characteristics identical to a `NoHeapRealtimeThread`$_{33}$ .
- `group` - A `ProcessingGroupParameters`$_{67}$ 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.

*Throws:*

`{@link` - IllegalArgumentException} if the initial memory area is in heap memory, and the `noheap` parameter is true.

---

# 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`$_{198}$ 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)
```
(`AsynchronouslyInterruptedException`$_{198}$)

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`$_{198}$ from within the `interruptAction()` method.

```
public void run(AsynchronouslyInterruptedException exception)
        throws AsynchronouslyInterruptedException
```
(`AsynchronouslyInterruptedException`$_{198}$)

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`$_{198}$ from within the `run()` method.

*Throws:*

`AsynchronouslyInterruptedException`$_{198}$

## 9.5 AsynchronouslyInterruptedException

*Declaration* :

```
public class AsynchronouslyInterruptedException extends
          java.lang.InterruptedException
```

*All Implemented Interfaces:* java.io.Serializable

*Direct Known Subclasses:* `Timed`$_{201}$

*Description* :

An special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a `RealtimeThread`$_{23}$ .

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()`$_{27}$ , or `this.fire()` is called, the `AsynchronouslyInterruptedException` is compared against any currently pending `AsynchronouslyInterruptedException` on the thread. If there is none, or if the depth of the `AsynchronouslyInterruptedException` is less than the currently pending `AsynchronouslyInterruptedException` —- i.e., it is targeted at a less deeply nested method call —- it becomes the currently pending interrupt. Otherwise, it is discarded.

If the current method is interruptible, the exception is thrown on the thread. Otherwise, it just remains pending until control returns to an interruptible method, at which point the `AsynchronouslyInterruptedException` is thrown. When an interrupt is caught, the caller should invoke the `happened()` method on the `AsynchronouslyInterruptedException` in which it is interested to see if it matches the pending `AsynchronouslyInterruptedException`. If so, the pending `AsynchronouslyInterruptedException` is cleared from the thread. Otherwise, it will continue to propagate outward.

`Thread.interrupt()` and `RealtimeThread.interrupt()` generate a system available generic `AsynchronouslyInterruptedException` which will always propagate outward through interruptible methods until the generic `AsynchronouslyInterruptedException` is identified and stopped. Other sources (e.g., `this.fire()` and `Timed`$_{201}$) 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`$_{197}$ `logic)`

Execute the `run()` method of the given `Interruptible`$_{197}$. This method may be on the stack in exactly one `RealtimeThread`$_{23}$. 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`$_{197}$ whose `run()` method will be called.

*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`$_{198}$ **`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`$_{198}$

*All Implemented Interfaces:* java.io.Serializable

*Description* :
Create a scope in a `RealtimeThread`$_{23}$ 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(HighResolutionTime148 time)
            throws IllegalArgumentException
```

Create an instance of `Timed` with a timer set to timeout. If the time is in the past the `AsynchronouslyInterruptedException`$_{198}$ 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(Interruptible197 logic)
```

Execute a timeout method. Starts the timer and executes the `run()` method of the given `Interruptible`$_{197}$ object.

*Overrides:* `public boolean doInterruptible(Interruptible`$_{197}$ `logic)`$_{199}$ in class `AsynchronouslyInterruptedException`$_{198}$

*Parameters:*

`logic` - Implements an `Interruptible`$_{197}$ `run()` method. If null nothing happens.

`public void` **`resetTime`**`(HighResolutionTime`$_{148}$ `time)`

To reschedule the timeout for the next invocation of `doInterruptible()`.

*Parameters:*

`time` - This can be an absolute time or a relative time. If null the timeout is not changed.

Chapter 10

# System and Options

This section contains classes that:

- Provide a common idiom for binding POSIX signals to instances of `AsyncEvent` when POSIX signals are available on the underlying platform.
- Provide a class that contains operations and semantics that affect the entire system.
- Provide the security semantics required by the additional features in the entirety of this specification, which are additional to those required by implementations of the Java Language Specification.

The `RealtimeSecurity` class provides security primarily for physical memory access.

## Semantics and Requirements

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

1. The POSIX signal handler class is required to be available when implementations of this specification execute on an underlying platform that provides POSIX signals or any subset of signals named with the POSIX names.
2. The RealtimeSecurity class is required.

## 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* :

```
public final class POSIXSignalHandler
```

*Description* :

Use instances of `AsyncEvent`181 to handle POSIX signals. Usage:

```
POSIXSignalHandler.addHandler(SIGINT, intHandler);
```

This class is required to be implemented only if the underlying operating system supports POSIX signals.

### 10.1.1 Fields

`public static final int` **`SIGABRT`**

Used by abort, replace SIGIOT in the future.

`public static final int` **`SIGALRM`**

Alarm clock.

`public static final int` **`SIGBUS`**

Bus error.

`public static final int` **`SIGCANCEL`**

Thread cancellation signal used by libthread.

`public static final int` **`SIGCHLD`**

Child status change alias (POSIX).

`public static final int` **`SIGCLD`**

Child status change.

`public static final int` **`SIGCONT`**

Stopped process has been continued.

`public static final int` **`SIGEMT`**

EMT instruction.

`public static final int` **`SIGFPE`**

Floating point exception.

`public static final int` **`SIGFREEZE`**

Special signal used by CPR.

`public static final int` **`SIGHUP`**

Hangup.

`public static final int` **`SIGILL`**

Illegal instruction (not reset when caught).

`public static final int` **`SIGINT`**

Interrupt (rubout).

`public static final int` **`SIGIO`**

Socket I/O possible (SIGPOLL alias).

`public static final int` **`SIGIOT`**

IOT instruction.

`public static final int` **`SIGKILL`**

Kill (cannot be caught or ignored).

`public static final int` **`SIGLOST`**

Resource lost (e.g., record-lock lost).

`public static final int` **`SIGLWP`**

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.

`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` **`SIGTTIN`**

Background tty read attempted.

`public static final int` **`SIGTTOU`**

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` **`SIGWAITING`**

Process's lwps are blocked.

`public static final int` **`SIGWINCH`**

Window size change.

`public static final int` **`SIGXCPU`**

Exceeded cpu limit.

`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`$_{183}$ `handler)`

Add the given `AsyncEventHandler`$_{183}$ to the list of handlers of the `AsyncEvent`$_{181}$ 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`$_{183}$ which will be scheduled when the given signal occurs.

`public static void` **`removeHandler`**`(int signal,`
`AsyncEventHandler`$_{183}$ `handler)`

Remove the given `AsyncEventHandler`$_{183}$ to the list of handlers of the `AsyncEvent`$_{181}$ 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`$_{183}$ which will be scheduled when the given signal occurs.

`public static void` **`setHandler`**`(int signal,`
`AsyncEventHandler`$_{183}$ `handler)`

Set the given `AsyncEventHandler`$_{183}$ as the handler of the `AsyncEvent`$_{181}$ 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`*183* which will be scheduled when the given signal occurs. If h is null then no handler will be associated with this (i.e., remove all handlers).

---

## 10.2 RealtimeSecurity

*Declaration* :

`public class RealtimeSecurity`

*Description* :

Security policy object for real-time specific issues. Primarily used to control access to physical memory.

### 10.2.1 Constructors

```
public RealtimeSecurity()
```

### 10.2.2 Methods

```
public void checkAccessPhysical()
            throws SecurityException
```

Check whether the application is allowed to access physical memory.

*Throws:*

`SecurityException` - the application doesn't have permission.

```
public void checkAccessPhysicalRange(long base,
            long size)
            throws SecurityException
```

Check whether the application is allowed to access physical memory within the specified range.

*Throws:*

`SecurityException` - the application doesn't have permission.

```
public void checkSetFilter()
```

```
                throws SecurityException
```

Check whether the application is allowed to set filter objects.

*Throws:*

`SecurityException` - the application doesn't have permission.

```
public void checkSetScheduler()
                throws SecurityException
```

Check whether the application is allowed to set the scheduler.

*Throws:*

`SecurityException` - the application doesn't have permission.

## 10.3 RealtimeSystem

*Declaration* :

```
public final class RealtimeSystem
```

*Description* :

`RealtimeSystem` provides a means for tuning the behavior of the implementation by specifying parameters such as the maximum number of locks that can be in use concurrently, and the monitor control policy. In addition, `RealtimeSystem` provides a mechanism for obtaining access to the security manager, garbage collector and scheduler, to make queries from them or to set parameters.

### 10.3.1 Fields

```
public static final byte BIG_ENDIAN
```

```
public static final byte BYTE_ORDER
```

```
public static final byte LITTLE_ENDIAN
```

### 10.3.2 Constructors

```
public RealtimeSystem()
```

### 10.3.3 Methods

`public static GarbageCollector`$_{132}$ **`currentGC`**`()`

Return a reference to the currently active garbage collector for the heap.

*Returns:* A `GarbageCollector`$_{132}$ 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`$_{209}$ **`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`$_{209}$ 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` 221 is thrown.

```
public static void setSecurityManager(RealtimeSecurity209
              manager)
```

Set a new real-time security manager.

*Parameters:*

`manager` - A `RealtimeSecurity` 209 object which will become the new security manager.

*Throws:*

`SecurityException` - Thrown if security manager has already been set.

Chapter 11

# Exceptions



This section contains classes that:

- Add additional exception classes required by the entirety of the other sections of this specification.
- Provide for the ability to asynchronously transfer the control of program logic.

## Semantics and Requirements

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

1. All classes in this section are required.
2. All exceptions, except `AsynchronouslyInterruptedException`, are required to have semantics exactly as those of their eventual superclass in the `java.*` hierarchy.
3. Instances of the class `AsynchronouslyInterruptedException` can be generated by execution of program logic and by internal virtual machine mechanisms that are asynchronous to the execution of program logic which is the target of the exception.
4. Program logic that exists in methods that throw `AsynchronouslyInterrupted-Exception` is subject to receiving an instance of `AsynchronouslyInterrupted-`

`Exception` at any time during execution.

## Rationale

The need for additional exceptions given the new semantics added by the other sections of this specification is obvious. That the specification attaches new, nontraditional, exception semantics to `AsynchronouslyInterruptedException` is, perhaps, not so obvious. However, after careful thought, and given our self-imposed directive that only well-defined code blocks would be subject to having their control asynchronously transferred, the chosen mechanism is logical.

## 11.1 DuplicateFilterException

*Declaration* :

```
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

```
public DuplicateFilterException()
```

```
public DuplicateFilterException(java.lang.String s)
```

*Parameters:*

`s` - Detail string

## 11.2 InaccessibleAreaException

*Declaration* :

```
public class InaccessibleAreaException extends
            java.lang.Exception
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :
The specified memory area is not above the current allocation context on the current thread scope stack.

### 11.2.1 Constructors

```
public InaccessibleAreaException()
```

A constructor for `InaccessibleAreaException`.

```
public InaccessibleAreaException(java.lang.String
            description)
```

A descriptive constructor for `InaccessibleAreaException`.

*Parameters:*
`description` - Description of the error.

---

## 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`*95* 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`$_{183}$ with a `ReleaseParameters`$_{54}$ type of `SporadicParameters` has `mitViolationExcept` behavior and the minimum interarrival time gets violated.

### 11.5.1 Constructors

```
public MITViolationException()
```

A constructor for `MITViolationException`.

```
public MITViolationException(java.lang.String
                description)
```

A descriptive constructor for `MITViolationException`.

*Parameters:*

`description` - Description of the error.

---

## 11.6 OffsetOutOfBoundsException

*Declaration* :

```
public class OffsetOutOfBoundsException extends
            java.lang.Exception
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :

Thrown if the constructor of an `ImmortalPhysicalMemory`$_{100}$, `LTPhysicalMemory`$_{106}$, `VTPhysicalMemory`$_{112}$, `RawMemoryAccess`$_{117}$, or `RawMemoryFloatAccess`$_{125}$ is given an invalid address.

### 11.6.1 Constructors

```
public OffsetOutOfBoundsException()
```

```
public OffsetOutOfBoundsException(java.lang.String
                description)
```

---

## 11.7 SizeOutOfBoundsException

*Declaration* :

```
public class SizeOutOfBoundsException extends
            java.lang.Exception
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :

Thrown if the constructor of an `ImmortalPhysicalMemory`$_{100}$, `LTPhysicalMemory`$_{106}$, `VTPhysicalMemory`$_{112}$, `RawMemoryAccess`$_{117}$, or

`RawMemoryFloatAccess`$_{125}$ 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`$_{117}$, `RawMemoryFloatAccess`$_{125}$, `ImmortalPhysicalMemory`$_{100}$, `LTPhysicalMemory`$_{106}$, `VTPhysicalMemory`$_{112}$

### 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* :

```
public class MemoryInUseException extends
            java.lang.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

```
public MemoryInUseException()
```

```
public MemoryInUseException(java.lang.String s)
```

*Parameters:*

`s` - Detail string

## 11.10 ScopedCycleException

*Declaration* :

```
public class ScopedCycleException extends
            java.lang.RuntimeException
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :

Thrown when a user tries to enter a `ScopedMemory`$_{84}$ that is already accessible (`ScopedMemory`$_{84}$ is present on stack) or when a user tries to create `ScopedMemory`$_{84}$ cycle spanning threads (tries to make cycle in the VM `ScopedMemory`$_{84}$ tree structure).

### 11.10.1 Constructors

public **ScopedCycleException**()

public **ScopedCycleException**(java.lang.String description)

---

## 11.11 UnknownHappeningException

*Declaration* :

```
public class UnknownHappeningException extends
            java.lang.RuntimeException
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :

Thrown when `bindTo()` is called with an illegal `happening`.

### 11.11.1 Constructors

public **UnknownHappeningException**()

public **UnknownHappeningException**(java.lang.String description)

---

## 11.12 IllegalAssignmentError

*Declaration* :

```
public class IllegalAssignmentError extends java.lang.Error
```

*All Implemented Interfaces:* java.io.Serializable

*Description* :

The exception thrown on an attempt to make an illegal assignment. For example, this will be thrown if logic attempts to assign a reference to an object in `ScopedMemory` to a field in an object in `ImmortalMemory`.

### 11.12.1 Constructors

public **IllegalAssignmentError**()

A constructor for `IllegalAssignmentError`.

```
public IllegalAssignmentError(java.lang.String
                description)
```

A descriptive constructor for `IllegalAssignmentError`.

*Parameters:*

`description` - Description of the error.

## 11.13 MemoryAccessError

*Declaration* :

`public class **MemoryAccessError** extends java.lang.Error`

*All Implemented Interfaces:* java.io.Serializable

*Description* :

This error is thrown on an attempt to refer to an object in an inaccessible `MemoryArea`$_{77}$. For example this will be thrown if logic in a `NoHeapRealtimeThread`$_{33}$ attempts to refer to an object in the traditional Java heap.

### 11.13.1 Constructors

```
public MemoryAccessError()
```

A constructor for `MemoryAccessError`.

```
public MemoryAccessError(java.lang.String description)
```

A descriptive constructor for `MemoryAccessError`.

*Parameters:*

`description` - Description of the error.

## 11.14 ResourceLimitError

*Declaration* :

`public class **ResourceLimitError** extends java.lang.Error`

*All Implemented Interfaces:* java.io.Serializable

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

### 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.

## ALMANAC LEGEND

The almanac presents classes and intefaces in alphabetic order, regardless of their package. Fields, methods and constructors are in alphabetic order in a single list.

This almanac is modeled after the style introduced by Patrick Chan in his excellent book *Java Developers Almanac*.

| ❶ RealtimeThread | | | ❷ javax.realtime |
|---|---|---|---|
| ❸ Object<br>➥ Thread<br>➥ RealtimeThread | | | ❹ Runnable<br>Schedulable |
| ❺ | ❻ | void | **addToFeasibility**() |
| | | RealtimeThread | **currentRealtimeThread**() |
| 1.3 | ❑ | Scheduler | **getScheduler**() |
| | ✻ | | **RealtimeThread**() |
| 1.3 | ✻ | | **RealtimeThread**(SchedulingParameters scheduling) |
| | ❑ | void | **sleep**(Clock clock, HighResolutionTime time) |
| | | ❼ | ❽ *throws* InterruptedException |

1. Name of the class, interface, nested class or nested interface. Interfaces are italic.
2. Name of the package containing the class or interface.
3. Inheritance hierarchy. In this example, `RealtimeThread` extends `Thread`, which extends `Object`.
4. Implemented interfaces. The interface is to the right of, and on the same line as, the class that implements it. In this example, `Thread` implements `Runnable`, and `RealtimeThread` implements `Schedulable`.
5. The first column above is for the value of the `@since` comment, which indicates the version in which the item was introduced.
6. The second column above is for the following icons. If the "protected" symbol does not appear, the member is public. (Private and package-private modifiers also have no symbols.) One symbol from each group can appear in this column.

| **Modifiers** | **Access Modifiers** | **Constructors and Fields** |
|---|---|---|
| ❍ abstract | ♦protected | ✻ constructor |
| ● final | | ✍ field |
| ❑ static | | |
| ■ static final | | |

7. Return type of a method or declared type of a field. Blank for constructors.
8. Name of the constructor, field or method. Nested classes are listed in 1, not here.

Chapter 12

# Almanac



## AbsoluteTime — javax.realtime

Object
➥HighResolutionTime — Comparable
➥AbsoluteTime

| | | |
|---|---|---|
| | AbsoluteTime | **absolute(Clock clock)** |
| | AbsoluteTime | **absolute(Clock clock, AbsoluteTime destination)** |
| ❇ | | **AbsoluteTime()** |
| ❇ | | **AbsoluteTime(AbsoluteTime time)** |
| ❇ | | **AbsoluteTime(java.util.Date date)** |
| ❇ | | **AbsoluteTime(long millis, int nanos)** |
| | AbsoluteTime | **add(long millis, int nanos)** |
| | AbsoluteTime | **add(long millis, int nanos, AbsoluteTime destination)** |
| ● | AbsoluteTime | **add(RelativeTime time)** |
| | AbsoluteTime | **add(RelativeTime time, AbsoluteTime destination)** |
| | java.util.Date | **getDate()** |
| | RelativeTime | **relative(Clock clock)** |
| | RelativeTime | **relative(Clock clock, AbsoluteTime destination)** |
| | void | **set(java.util.Date date)** |
| ● | RelativeTime | **subtract(AbsoluteTime time)** |
| ● | RelativeTime | **subtract(AbsoluteTime time, RelativeTime destination)** |

| | | |
|---|---|---|
| ● | AbsoluteTime | **subtract(RelativeTime time)** |
| | AbsoluteTime | **subtract(RelativeTime time, AbsoluteTime destination)** |
| | String | **toString()** |

## AperiodicParameters — javax.realtime

Object
➥ReleaseParameters
➥AperiodicParameters

| | | |
|---|---|---|
| ❇ | | **AperiodicParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)** |
| | boolean | **setIfFeasible(RelativeTime cost, RelativeTime deadline)** |

## AsyncEvent — javax.realtime

Object
➥AsyncEvent

| | | |
|---|---|---|
| | void | **addHandler(AsyncEventHandler handler)** |
| ❇ | | **AsyncEvent()** |
| | void | **bindTo(String happening)** ***throws*** **UnknownHappeningException** |
| | ReleaseParameters | **createReleaseParameters()** |
| | void | **fire()** |
| | boolean | **handledBy(AsyncEventHandler handler)** |
| | void | **removeHandler(AsyncEventHandler handler)** |
| | void | **setHandler(AsyncEventHandler handler)** |
| | void | **unbindTo(String happening)** ***throws*** **UnknownHappeningException** |

## AsyncEventHandler — javax.realtime

Object
➥AsyncEventHandler — Schedulable

| | | |
|---|---|---|
| | boolean | **addIfFeasible()** |
| | boolean | **addToFeasibility()** |
| ❇ | | **AsyncEventHandler()** |
| ❇ | | **AsyncEventHandler(boolean nonheap)** |
| ❇ | | **AsyncEventHandler(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(ProcessingGroupParameters group)** |
| | void | **setReleaseParameters(ReleaseParameters release)** |
| | boolean | **setReleaseParametersIfFeasible(ReleaseParameters release)** |

| | | |
|---|---|---|
| | void | **setScheduler(Scheduler scheduler)** ***throws*** **IllegalThreadStateException** |
| | void | **setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup)** ***throws*** **IllegalThreadStateException** |
| | void | **setSchedulingParameters(SchedulingParameters scheduling)** |
| | boolean | **setSchedulingParametersIfFeasible(SchedulingParameters sched)** |

## AsynchronouslyInterruptedException — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Exception
➥InterruptedException
➥AsynchronouslyInterruptedException

| | | |
|---|---|---|
| ❇ | | **AsynchronouslyInterruptedException()** |
| | boolean | **disable()** |
| | boolean | **doInterruptible(Interruptible logic)** |
| | boolean | **enable()** |
| | boolean | **fire()** |
| ❏ | AsynchronouslyInterruptedException | **getGeneric()** |
| | boolean | **happened(boolean propagate)** |
| | boolean | **isEnabled()** |
| ❏ | void | **propagate()** |

## BoundAsyncEventHandler — javax.realtime

Object
➥AsyncEventHandler — Schedulable
➥BoundAsyncEventHandler

| | | |
|---|---|---|
| ❇ | | **BoundAsyncEventHandler()** |
| ❇ | | **BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean nonheap, Runnable logic)** |

| Clock | javax.realtime |
|---|---|

Object
➥Clock

| | | |
|---|---|---|
| ✻ | | **Clock()** |
| ❏ | Clock | **getRealtimeClock()** |
| ❍ | RelativeTime | **getResolution()** |
| | AbsoluteTime | **getTime()** |
| ❍ | void | **getTime(AbsoluteTime time)** |
| ❍ | void | **setResolution(RelativeTime resolution)** |

| DuplicateFilterException | javax.realtime |
|---|---|

Object
➥Throwable — java.io.Serializable
➥Exception
➥DuplicateFilterException

| | | |
|---|---|---|
| ✻ | | **DuplicateFilterException()** |
| ✻ | | **DuplicateFilterException(String s)** |

| GarbageCollector | javax.realtime |
|---|---|

Object
➥GarbageCollector

| | | |
|---|---|---|
| ✻ | | **GarbageCollector()** |
| ❍ | RelativeTime | **getPreemptionLatency()** |

| HeapMemory | javax.realtime |
|---|---|

Object
➥MemoryArea
➥HeapMemory

| | | |
|---|---|---|
| ❏ | HeapMemory | **instance()** |
| | long | **memoryConsumed()** |
| | long | **memoryRemaining()** |

## HighResolutionTime — javax.realtime

Object
➥HighResolutionTime — Comparable

| | | |
|---|---|---|
| ❍ | AbsoluteTime | **absolute(Clock clock)** |
| ❍ | AbsoluteTime | **absolute(Clock clock, AbsoluteTime dest)** |
| | int | **compareTo(HighResolutionTime time)** |
| | int | **compareTo(Object object)** |
| | boolean | **equals(HighResolutionTime time)** |
| | boolean | **equals(Object object)** |
| ● | long | **getMilliseconds()** |
| ● | int | **getNanoseconds()** |
| | int | **hashCode()** |
| ❍ | RelativeTime | **relative(Clock clock)** |
| ❍ | RelativeTime | **relative(Clock clock, HighResolutionTime time)** |
| | void | **set(HighResolutionTime time)** |
| | void | **set(long millis)** |
| | void | **set(long millis, int nanos)** |
| ❑ | void | **waitForObject(Object target, HighResolutionTime time) *throws* InterruptedException** |

## IllegalAssignmentError — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Error
➥IllegalAssignmentError

| | | |
|---|---|---|
| ❇ | | **IllegalAssignmentError()** |
| ❇ | | **IllegalAssignmentError(String description)** |

## ImmortalMemory — javax.realtime

Object
➥MemoryArea
➥ImmortalMemory

| | | |
|---|---|---|
| ❑ | ImmortalMemory | **instance()** |

## ImmortalPhysicalMemory javax.realtime

Object
➥MemoryArea
➥ImmortalPhysicalMemory

- ❇ **ImmortalPhysicalMemory(Object type, long size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
- ❇ **ImmortalPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
- ❇ **ImmortalPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
- ❇ **ImmortalPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
- ❇ **ImmortalPhysicalMemory(Object type, long base, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
- ❇ **ImmortalPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException**
- ❇ **ImmortalPhysicalMemory(Object type, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**
- ❇ **ImmortalPhysicalMemory(Object type, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException**

## ImportanceParameters — javax.realtime

Object
➥SchedulingParameters
➥PriorityParameters
➥ImportanceParameters

| | | |
|---|---|---|
| | int | **getImportance()** |
| ❇ | | **ImportanceParameters(int priority, int importance)** |
| | void | **setImportance(int importance)** |
| | String | **toString()** |

## InaccessibleAreaException — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Exception
➥InaccessibleAreaException

| | | |
|---|---|---|
| ❇ | | **InaccessibleAreaException()** |
| ❇ | | **InaccessibleAreaException(String description)** |

## *Interruptible* — javax.realtime

Interruptible

| | | |
|---|---|---|
| | void | **interruptAction(AsynchronouslyInterruptedException exception)** |
| | void | **run(AsynchronouslyInterruptedException exception) *throws* AsynchronouslyInterruptedException** |

## LTMemory — javax.realtime

Object
➥MemoryArea
➥ScopedMemory
➥LTMemory

| | | |
|---|---|---|
| | long | **getMaximumSize()** |
| ❇ | | **LTMemory(long initialSizeInBytes, long maxSizeInBytes)** |
| ❇ | | **LTMemory(long initialSizeInBytes, long maxSizeInBytes, Runnable logic)** |

| | | |
|---|---|---|
| ❇ | | **LTMemory(SizeEstimator initial, SizeEstimator maximum)** |
| ❇ | | **LTMemory(SizeEstimator initial, SizeEstimator maximum, Runnable logic)** |
| | String | **toString()** |

## LTPhysicalMemory — javax.realtime

Object
➥MemoryArea
➥ScopedMemory
➥LTPhysicalMemory

| | | |
|---|---|---|
| ❇ | | **LTPhysicalMemory(Object type, long size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **LTPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **LTPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **LTPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **LTPhysicalMemory(Object type, long base, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **LTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |

| | | |
|---|---|---|
| ❇ | | **LTPhysicalMemory(Object type, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **LTPhysicalMemory(Object type, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| | String | **toString()** |

## MemoryAccessError — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Error
➥MemoryAccessError

| | | |
|---|---|---|
| ❇ | | **MemoryAccessError()** |
| ❇ | | **MemoryAccessError(String description)** |

## MemoryArea — javax.realtime

Object
➥MemoryArea

| | | |
|---|---|---|
| | void | **enter() *throws* ScopedCycleException** |
| | void | **enter(Runnable logic) *throws* ScopedCycleException** |
| | void | **executeInArea(Runnable logic) *throws* InaccessibleAreaException** |
| ❏ | MemoryArea | **getMemoryArea(Object object)** |
| ❇♦ | | **MemoryArea(long sizeInBytes)** |
| ❇♦ | | **MemoryArea(long sizeInBytes, Runnable logic)** |
| ❇♦ | | **MemoryArea(SizeEstimator size)** |
| ❇♦ | | **MemoryArea(SizeEstimator size, Runnable logic)** |
| | long | **memoryConsumed()** |
| | long | **memoryRemaining()** |
| | Object | **newArray(Class type, int number) *throws* IllegalAccessException, InstantiationException** |
| | Object | **newInstance(Class type) *throws* IllegalAccessException, InstantiationException** |
| | Object | **newInstance(reflect.Constructor c, Object[] args) *throws* IllegalAccessException, InstantiationException** |
| | long | **size()** |

## MemoryInUseException — javax.realtime

Object
 ➥Throwable — java.io.Serializable
  ➥Exception
   ➥RuntimeException
    ➥MemoryInUseException

| | | |
|---|---|---|
| ❇ | | **MemoryInUseException()** |
| ❇ | | **MemoryInUseException(String s)** |

## MemoryParameters — javax.realtime

Object
 ➥MemoryParameters

| | | |
|---|---|---|
| | long | **getAllocationRate()** |
| | long | **getMaxImmortal()** |
| | long | **getMaxMemoryArea()** |
| ❇ | | **MemoryParameters(long maxMemoryArea, long maxImmortal)** ***throws*** **IllegalArgumentException** |
| ❇ | | **MemoryParameters(long maxMemoryArea, long maxImmortal, long allocationRate)** ***throws*** **IllegalArgumentException** |
| ✍■ | long | **NO_MAX** |
| | void | **setAllocationRate(long allocationRate)** |
| | boolean | **setAllocationRateIfFeasible(int allocationRate)** |
| | boolean | **setMaxImmortalIfFeasible(long maximum)** |
| | boolean | **setMaxMemoryAreaIfFeasible(long maximum)** |

## MemoryScopeException — javax.realtime

Object
 ➥Throwable — java.io.Serializable
  ➥Exception
   ➥MemoryScopeException

| | | |
|---|---|---|
| ❇ | | **MemoryScopeException()** |
| ❇ | | **MemoryScopeException(String description)** |

## MemoryTypeConflictException — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Exception
➥MemoryTypeConflictException

| | | |
|---|---|---|
| ❇ | | **MemoryTypeConflictException()** |
| ❇ | | **MemoryTypeConflictException(String s)** |

## MITViolationException — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Exception
➥MITViolationException

| | | |
|---|---|---|
| ❇ | | **MITViolationException()** |
| ❇ | | **MITViolationException(String description)** |

## MonitorControl — javax.realtime

Object
➥MonitorControl

| | | |
|---|---|---|
| ❏ | MonitorControl | **getMonitorControl()** |
| ❏ | MonitorControl | **getMonitorControl(Object monitor)** |
| ❇ | | **MonitorControl()** |
| ❏ | void | **setMonitorControl(MonitorControl policy)** |
| ❏ | void | **setMonitorControl(Object monitor, MonitorControl monCtl)** |

## NoHeapRealtimeThread — javax.realtime

Object
 ➥Thread — Runnable
  ➥RealtimeThread — Schedulable
   ➥NoHeapRealtimeThread

| | | |
|---|---|---|
| ❇ | | **NoHeapRealtimeThread(SchedulingParameters sp, MemoryArea ma)** ***throws*** **IllegalArgumentException** |
| ❇ | | **NoHeapRealtimeThread(SchedulingParameters sp, ReleaseParameters rp, MemoryArea ma)** ***throws*** **IllegalArgumentException** |
| ❇ | | **NoHeapRealtimeThread(SchedulingParameters sp, ReleaseParameters rp, MemoryParameters mp, MemoryArea ma, ProcessingGroupParameters group, Runnable logic)** ***throws*** **IllegalArgumentException** |
| | void | **start()** |

## OffsetOutOfBoundsException — javax.realtime

Object
 ➥Throwable — java.io.Serializable
  ➥Exception
   ➥OffsetOutOfBoundsException

| | |
|---|---|
| ❇ | **OffsetOutOfBoundsException()** |
| ❇ | **OffsetOutOfBoundsException(String description)** |

## OneShotTimer — javax.realtime

Object
 ➥AsyncEvent
  ➥Timer
   ➥OneShotTimer

| | |
|---|---|
| ❇ | **OneShotTimer(HighResolutionTime time, AsyncEventHandler handler)** |
| ❇ | **OneShotTimer(HighResolutionTime start, Clock clock, AsyncEventHandler handler)** |

## PeriodicParameters — javax.realtime

Object
  ➥ReleaseParameters
    ➥PeriodicParameters

| | | |
|---|---|---|
| | RelativeTime | **getPeriod()** |
| | HighResolutionTime | **getStart()** |
| ❇ | | **PeriodicParameters(HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)** |
| | boolean | **setIfFeasible(RelativeTime period, RelativeTime cost, RelativeTime deadline)** |
| | void | **setPeriod(RelativeTime p)** |
| | void | **setStart(HighResolutionTime s)** |

## PeriodicTimer — javax.realtime

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 — javax.realtime

Object
  ➥PhysicalMemoryManager

| | | |
|---|---|---|
| ✍■ | String | **ALIGNED** |
| ✍■ | String | **BYTESWAP** |
| ✍■ | String | **DMA** |
| ❏ | boolean | **isRemovable(long address, long size)** |

| | | |
|---|---|---|
| ❏ | boolean | **isRemoved(long address, long size)** |
| ❏ | void | **onInsertion(long base, long size, AsyncEventHandler aeh)** |
| ❏ | void | **onRemoval(long base, long size, AsyncEventHandler aeh)** |
| ■ | void | **registerFilter(Object name, PhysicalMemoryTypeFilter filter)** ***throws*** **DuplicateFilterException, IllegalArgumentException** |
| ■ | void | **removeFilter(Object name)** |
| ✍■ | String | **SHARED** |

## ***PhysicalMemoryTypeFilter*** — javax.realtime

PhysicalMemoryTypeFilter

| | | |
|---|---|---|
| | boolean | **contains(long base, long size)** |
| | long | **find(long base, long size)** |
| | int | **getVMAttributes()** |
| | int | **getVMFlags()** |
| | void | **initialize(long base, long vBase, long size)** |
| | boolean | **isPresent(long base, long size)** |
| | boolean | **isRemovable()** |
| | void | **onInsertion(long base, long size, AsyncEventHandler aeh)** |
| | void | **onRemoval(long base, long size, AsyncEventHandler aeh)** |
| | long | **vFind(long base, long size)** |

## **POSIXSignalHandler** — javax.realtime

Object
  ➥POSIXSignalHandler

| | | |
|---|---|---|
| ❏ | void | **addHandler(int signal, AsyncEventHandler handler)** |
| ✻ | | **POSIXSignalHandler()** |
| ❏ | void | **removeHandler(int signal, AsyncEventHandler handler)** |
| ❏ | void | **setHandler(int signal, AsyncEventHandler handler)** |
| ✍■ | int | **SIGABRT** |
| ✍■ | int | **SIGALRM** |
| ✍■ | int | **SIGBUS** |
| ✍■ | int | **SIGCANCEL** |
| ✍■ | int | **SIGCHLD** |
| ✍■ | int | **SIGCLD** |

int **SIGCONT**
int **SIGEMT**
int **SIGFPE**
int **SIGFREEZE**
int **SIGHUP**
int **SIGILL**
int **SIGINT**
int **SIGIO**
int **SIGIOT**
int **SIGKILL**
int **SIGLOST**
int **SIGLWP**
int **SIGPIPE**
int **SIGPOLL**
int **SIGPROF**
int **SIGPWR**
int **SIGQUIT**
int **SIGSEGV**
int **SIGSTOP**
int **SIGSYS**
int **SIGTERM**
int **SIGTHAW**
int **SIGTRAP**
int **SIGTSTP**
int **SIGTTIN**
int **SIGTTOU**
int **SIGURG**
int **SIGUSR1**
int **SIGUSR2**
int **SIGVTALRM**
int **SIGWAITING**
int **SIGWINCH**
int **SIGXCPU**
int **SIGXFSZ**

## PriorityCeilingEmulation — javax.realtime

Object
➥MonitorControl
➥PriorityCeilingEmulation

| | | |
|---|---|---|
| | int | **getDefaultCeiling()** |
| ❇ | | **PriorityCeilingEmulation(int ceiling)** |

## PriorityInheritance — javax.realtime

Object
➥MonitorControl
➥PriorityInheritance

| | | |
|---|---|---|
| ❏ | PriorityInheritance | **instance()** |
| ❇ | | **PriorityInheritance()** |

## PriorityParameters — javax.realtime

Object
➥SchedulingParameters
➥PriorityParameters

| | | |
|---|---|---|
| | int | **getPriority()** |
| ❇ | | **PriorityParameters(int priority)** |
| | void | **setPriority(int priority)** ***throws*** **IllegalArgumentException** |
| | String | **toString()** |

## PriorityScheduler — javax.realtime

Object
➥Scheduler
➥PriorityScheduler

| | | |
|---|---|---|
| ♦ | 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()** |

| | | |
|---|---|---|
| ❏ | PriorityScheduler | **instance()** |
| | boolean | **isFeasible()** |
| ✍■ | int | **MAX_PRIORITY** |
| ✍■ | int | **MIN_PRIORITY** |
| ❇♦ | | **PriorityScheduler()** |
| ♦ | boolean | **removeFromFeasibility(Schedulable schedulable)** |
| | boolean | **setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)** |
| | boolean | **setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)** |

## ProcessingGroupParameters — javax.realtime

Object
➥ProcessingGroupParameters

| | | |
|---|---|---|
| | RelativeTime | **getCost()** |
| | AsyncEventHandler | **getCostOverrunHandler()** |
| | RelativeTime | **getDeadline()** |
| | AsyncEventHandler | **getDeadlineMissHandler()** |
| | RelativeTime | **getPeriod()** |
| | HighResolutionTime | **getStart()** |
| ❇ | | **ProcessingGroupParameters(HighResolutionTime start, RelativeTime period, RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)** |
| | void | **setCost(RelativeTime cost)** |
| | void | **setCostOverrunHandler(AsyncEventHandler handler)** |
| | void | **setDeadline(RelativeTime deadline)** |
| | void | **setDeadlineMissHandler(AsyncEventHandler handler)** |
| | boolean | **setIfFeasible(RelativeTime period, RelativeTime cost, RelativeTime deadline)** |
| | void | **setPeriod(RelativeTime period)** |
| | void | **setStart(HighResolutionTime start)** |

| **RationalTime** | **javax.realtime** |
|---|---|

Object
➥HighResolutionTime　　　　Comparable
　➥RelativeTime
　　➥RationalTime

| | | |
|---|---|---|
| | AbsoluteTime | **absolute(Clock clock, AbsoluteTime destination)** |
| | void | **addInterarrivalTo(AbsoluteTime destination)** |
| | int | **getFrequency()** |
| | RelativeTime | **getInterarrivalTime()** |
| | RelativeTime | **getInterarrivalTime(RelativeTime dest)** |
| ❇ | | **RationalTime(int frequency)** |
| ❇ | | **RationalTime(int frequency, long millis, int nanos)** ***throws*** **IllegalArgumentException** |
| ❇ | | **RationalTime(int frequency, RelativeTime interval)** ***throws*** **IllegalArgumentException** |
| | void | **set(long millis, int nanos)** ***throws*** **IllegalArgumentException** |
| | void | **setFrequency(int frequency)** ***throws*** **ArithmeticException** |

| **RawMemoryAccess** | **javax.realtime** |
|---|---|

Object
➥RawMemoryAccess

| | | |
|---|---|---|
| | byte | **getByte(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **getBytes(long offset, byte[] bytes, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | int | **getInt(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **getInts(long offset, int[] ints, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | long | **getLong(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **getLongs(long offset, long[] longs, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | long | **getMappedAddress()** |

| | | |
|---|---|---|
| | short | **getShort(long offset)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **getShorts(long offset, short[] shorts, int low, int number)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | long | **map()** |
| | long | **map(long base)** |
| | long | **map(long base, long size)** |
| ❇ | | **RawMemoryAccess(Object type, long size)**<br>***throws*** **SecurityException, OffsetOutOfBound-sException, SizeOutOfBoundsException, Unsup-portedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **RawMemoryAccess(Object type, long base, long size)**<br>***throws*** **SecurityException, OffsetOutOfBound-sException, SizeOutOfBoundsException, Unsup-portedPhysicalMemoryException, MemoryTypeConflictException** |
| | void | **setByte(long offset, byte value)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setBytes(long offset, byte[] bytes, int low, int number)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setInt(long offset, int value)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setInts(long offset, int[] ints, int low, int number)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setLong(long offset, long value)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setLongs(long offset, long[] longs, int low, int number)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setShort(long offset, short value)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **setShorts(long offset, short[] shorts, int low, int number)**<br>***throws*** **OffsetOutOfBoundsException, Size-OutOfBoundsException** |
| | void | **unmap()** |

## RawMemoryFloatAccess — javax.realtime

Object
➥RawMemoryAccess
➥RawMemoryFloatAccess

| | | |
|---|---|---|
| | double | **getDouble(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **getDoubles(long offset, double[] doubles, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | float | **getFloat(long offset)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **getFloats(long offset, float[] floats, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| ❇ | | **RawMemoryFloatAccess(Object type, long size)** ***throws*** **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **RawMemoryFloatAccess(Object type, long base, long size)** ***throws*** **SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| | void | **setDouble(long offset, double value)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **setDoubles(long offset, double[] doubles, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **setFloat(long offset, float value)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |
| | void | **setFloats(long offset, float[] floats, int low, int number)** ***throws*** **OffsetOutOfBoundsException, SizeOutOfBoundsException** |

## RealtimeSecurity — javax.realtime

Object
  ➥RealtimeSecurity

| | | |
|---|---|---|
| | void | **checkAccessPhysical()** ***throws*** **SecurityException** |
| | void | **checkAccessPhysicalRange(long base, long size)** ***throws*** **SecurityException** |
| | void | **checkSetFilter()** ***throws*** **SecurityException** |
| | void | **checkSetScheduler()** ***throws*** **SecurityException** |
| ❊ | | **RealtimeSecurity()** |

## RealtimeSystem — javax.realtime

Object
  ➥RealtimeSystem

| | | |
|---|---|---|
| ✍■ | byte | **BIG_ENDIAN** |
| ✍■ | byte | **BYTE_ORDER** |
| ❑ | GarbageCollector | **currentGC()** |
| ❑ | int | **getConcurrentLocksUsed()** |
| ❑ | int | **getMaximumConcurrentLocks()** |
| ❑ | RealtimeSecurity | **getSecurityManager()** |
| ✍■ | byte | **LITTLE_ENDIAN** |
| ❊ | | **RealtimeSystem()** |
| ❑ | void | **setMaximumConcurrentLocks(int numLocks)** |
| ❑ | void | **setMaximumConcurrentLocks(int number, boolean hard)** |
| ❑ | void | **setSecurityManager(RealtimeSecurity manager)** |

## RealtimeThread — javax.realtime

Object
  ➥Thread — Runnable
    ➥RealtimeThread — Schedulable

| | | |
|---|---|---|
| | boolean | **addIfFeasible()** |
| | boolean | **addToFeasibility()** |
| ❑ | RealtimeThread | **currentRealtimeThread()** ***throws*** **ClassCastException** |
| | void | **deschedulePeriodic()** |
| ❑ | MemoryArea | **getCurrentMemoryArea()** |
| ❑ | int | **getInitialMemoryAreaIndex()** |
| ❑ | int | **getMemoryAreaStackDepth()** |

| | | |
|---|---|---|
| | MemoryParameters | **getMemoryParameters()** |
| ❑ | MemoryArea | **getOuterMemoryArea(int index)** |
| | ProcessingGroupParameters | **getProcessingGroupParameters()** |
| | ReleaseParameters | **getReleaseParameters()** |
| | Scheduler | **getScheduler()** |
| | SchedulingParameters | **getSchedulingParameters()** |
| | void | **interrupt()** |
| ✻ | | **RealtimeThread()** |
| ✻ | | **RealtimeThread(SchedulingParameters scheduling)** |
| ✻ | | **RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release)** |
| ✻ | | **RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, Runnable logic)** |
| | boolean | **removeFromFeasibility()** |
| | void | **schedulePeriodic()** |
| | boolean | **setIfFeasible(ReleaseParameters release, MemoryParameters memory)** |
| | boolean | **setIfFeasible(ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)** |
| | boolean | **setIfFeasible(ReleaseParameters release, ProcessingGroupParameters group)** |
| | void | **setMemoryParameters(MemoryParameters parameters ) *throws* IllegalThreadStateException** |
| | boolean | **setMemoryParametersIfFeasible(MemoryParameters 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** |

| | | |
|---|---|---|
| | void | **setSchedulingParameters(SchedulingParameters scheduling)** ***throws*** **IllegalThreadStateException** |
| | boolean | **setSchedulingParametersIfFeasible(SchedulingParameters scheduling)** |
| ❏ | void | **sleep(Clock clock, HighResolutionTime time)** ***throws*** **InterruptedException** |
| ❏ | void | **sleep(HighResolutionTime time)** ***throws*** **InterruptedException** |
| | void | **start()** |
| | boolean | **waitForNextPeriod()** ***throws*** **IllegalThreadStateException** |

## RelativeTime — javax.realtime

Object
➥HighResolutionTime — Comparable
➥RelativeTime

| | | |
|---|---|---|
| | AbsoluteTime | **absolute(Clock clock)** |
| | AbsoluteTime | **absolute(Clock clock, AbsoluteTime destination)** |
| | RelativeTime | **add(long millis, int nanos)** |
| | RelativeTime | **add(long millis, int nanos, RelativeTime destination)** |
| ● | RelativeTime | **add(RelativeTime time)** |
| | RelativeTime | **add(RelativeTime time, RelativeTime destination)** |
| | void | **addInterarrivalTo(AbsoluteTime destination)** |
| | RelativeTime | **getInterarrivalTime()** |
| | RelativeTime | **getInterarrivalTime(RelativeTime destination)** |
| | RelativeTime | **relative(Clock clock)** |
| | RelativeTime | **relative(Clock clock, RelativeTime destination)** |
| ❇ | | **RelativeTime()** |
| ❇ | | **RelativeTime(long millis, int nanos)** |
| ❇ | | **RelativeTime(RelativeTime time)** |
| ● | RelativeTime | **subtract(RelativeTime time)** |
| | RelativeTime | **subtract(RelativeTime time, RelativeTime destination)** |
| | String | **toString()** |

## ReleaseParameters — javax.realtime

Object
➥ReleaseParameters

| | | |
|---|---|---|
| | RelativeTime | **getCost()** |
| | AsyncEventHandler | **getCostOverrunHandler()** |
| | RelativeTime | **getDeadline()** |

| | | |
|---|---|---|
| | AsyncEventHandler | **getDeadlineMissHandler()** |
| ❇♦ | | **ReleaseParameters()** |
| ❇♦ | | **ReleaseParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler overrunHandler, AsyncEventHandler missHandler)** |
| | void | **setCost(RelativeTime cost)** |
| | void | **setCostOverrunHandler(AsyncEventHandler handler)** |
| | void | **setDeadline(RelativeTime deadline)** |
| | void | **setDeadlineMissHandler(AsyncEventHandler handler)** |
| | boolean | **setIfFeasible(RelativeTime cost, RelativeTime deadline)** |

## ResourceLimitError — javax.realtime

Object
➥Throwable — java.io.Serializable
➥Error
➥ResourceLimitError

| | | |
|---|---|---|
| ❇ | | **ResourceLimitError()** |
| ❇ | | **ResourceLimitError(String description)** |

## *Schedulable* — javax.realtime

Schedulable — Runnable

| | | |
|---|---|---|
| | boolean | **addToFeasibility()** |
| | MemoryParameters | **getMemoryParameters()** |
| | ProcessingGroupParameters | **getProcessingGroupParameters()** |
| | ReleaseParameters | **getReleaseParameters()** |
| | Scheduler | **getScheduler()** |
| | SchedulingParameters | **getSchedulingParameters()** |
| | boolean | **removeFromFeasibility()** |
| | void | **setMemoryParameters(MemoryParameters memory)** |
| | boolean | **setMemoryParametersIfFeasible(MemoryParameters memParam)** |
| | void | **setProcessingGroupParameters(ProcessingGroupParameters groupParameters)** |
| | boolean | **setProcessingGroupParametersIfFeasible(ProcessingGroupParameters groupParameters)** |
| | void | **setReleaseParameters(ReleaseParameters release)** |
| | boolean | **setReleaseParametersIfFeasible(ReleaseParameters release)** |

| | | |
|---|---|---|
| | void | **setScheduler(Scheduler scheduler)** *throws* **IllegalThreadStateException** |
| | void | **setScheduler(Scheduler scheduler, SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memoryParameters, ProcessingGroupParameters processingGroup)** *throws* **IllegalThreadStateException** |
| | void | **setSchedulingParameters(SchedulingParameters scheduling)** |
| | boolean | **setSchedulingParametersIfFeasible(SchedulingParameters scheduling)** |

## Scheduler — javax.realtime

Object
➥Scheduler

| | | |
|---|---|---|
| ❍✦ | boolean | **addToFeasibility(Schedulable schedulable)** |
| ❍ | void | **fireSchedulable(Schedulable schedulable)** |
| ❏ | Scheduler | **getDefaultScheduler()** |
| ❍ | String | **getPolicyName()** |
| ❍ | boolean | **isFeasible()** |
| ❍✦ | boolean | **removeFromFeasibility(Schedulable schedulable)** |
| ❊✦ | | **Scheduler()** |
| ❏ | void | **setDefaultScheduler(Scheduler scheduler)** |
| | boolean | **setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory)** |
| | boolean | **setIfFeasible(Schedulable schedulable, ReleaseParameters release, MemoryParameters memory, ProcessingGroupParameters group)** |

## SchedulingParameters — javax.realtime

Object
➥SchedulingParameters

| | | |
|---|---|---|
| ❊ | | **SchedulingParameters()** |

## ScopedCycleException — javax.realtime

Object
➥Throwable java.io.Serializable

➥Exception
➥RuntimeException
➥ScopedCycleException

| | | |
|---|---|---|
| ❇ | | **ScopedCycleException()** |
| ❇ | | **ScopedCycleException(String description)** |

## ScopedMemory — javax.realtime

Object
➥MemoryArea
➥ScopedMemory

| | | |
|---|---|---|
| | void | **enter()** ***throws*** **ScopedCycleException** |
| | void | **enter(Runnable logic)** ***throws*** **ScopedCycleException** |
| | long | **getMaximumSize()** |
| | Object | **getPortal()** |
| | int | **getReferenceCount()** |
| | void | **join()** ***throws*** **InterruptedException** |
| | void | **join(HighResolutionTime time)** ***throws*** **InterruptedException** |
| | void | **joinAndEnter()** ***throws*** **InterruptedException, 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()** |

## SizeEstimator — javax.realtime

Object
➥SizeEstimator

| | | |
|---|---|---|
| | long | **getEstimate()** |
| | void | **reserve(Class c, int n)** |
| | void | **reserve(SizeEstimator s)** |
| | void | **reserve(SizeEstimator s, int n)** |
| ❇ | | **SizeEstimator()** |

## SizeOutOfBoundsException — javax.realtime

Object
➥Throwable ◆ java.io.Serializable
➥Exception
➥SizeOutOfBoundsException

| | | |
|---|---|---|
| ❇ | | **SizeOutOfBoundsException()** |
| ❇ | | **SizeOutOfBoundsException(String description)** |

## SporadicParameters — javax.realtime

Object
➥ReleaseParameters
➥AperiodicParameters
➥SporadicParameters

| | | |
|---|---|---|
| ✍■ | String | **arrivalTimeQueueOverflowExcept** |
| ✍■ | String | **arrivalTimeQueueOverflowIgnore** |
| ✍■ | String | **arrivalTimeQueueOverflowReplace** |
| ✍■ | String | **arrivalTimeQueueOverflowSave** |
| | String | **getArrivalTimeQueueOverflowBehavior()** |
| | String | **getArrivalTimeQueueOverflowBehavior()** |
| | int | **getInitialArrivalTimeQueueLength()** |
| | int | **getInitialArrivalTimeQueueLength()** |
| | RelativeTime | **getMinimumInterarrival()** |
| | String | **getMitViolationBehavior()** |
| | String | **getMitViolationBehavior()** |
| ✍■ | String | **mitViolationExcept** |
| ✍■ | String | **mitViolationIgnore** |
| ✍■ | String | **mitViolationReplace** |
| ✍■ | String | **mitViolationSave** |

| | | | |
|---|---|---|---|
| | 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)** | |

## ThrowBoundaryError — javax.realtime

Object
  ➥Throwable — java.io.Serializable
    ➥Error
      ➥ThrowBoundaryError

| | |
|---|---|
| ❇ | **ThrowBoundaryError()** |
| ❇ | **ThrowBoundaryError(String description)** |

## Timed — javax.realtime

Object
  ➥Throwable — java.io.Serializable
    ➥Exception
      ➥InterruptedException
        ➥AsynchronouslyInterruptedException
          ➥Timed

| | | |
|---|---|---|
| | boolean | **doInterruptible(Interruptible logic)** |
| | void | **resetTime(HighResolutionTime time)** |
| ❇ | | **Timed(HighResolutionTime time)** ***throws*** **IllegalArgumentException** |

## Timer — javax.realtime

Object
➥AsyncEvent
➥Timer

| | | |
|---|---|---|
| | ReleaseParameters | **createReleaseParameters()** |
| | void | **destroy()** |
| | void | **disable()** |
| | void | **enable()** |
| | Clock | **getClock()** |
| | AbsoluteTime | **getFireTime()** |
| | boolean | **isRunning()** |
| | void | **reschedule(HighResolutionTime time)** |
| | void | **start()** |
| | boolean | **stop()** |
| ❇♦ | | **Timer(HighResolutionTime t, Clock c, AsyncEventHandler handler)** |

## UnknownHappeningException — javax.realtime

Object
➥Throwable    java.io.Serializable
➥Exception
➥RuntimeException
➥UnknownHappeningException

| | |
|---|---|
| ❇ | **UnknownHappeningException()** |
| ❇ | **UnknownHappeningException(String description)** |

## UnsupportedPhysicalMemoryEx- ception — javax.realtime

Object
➥Throwable    java.io.Serializable
➥Exception
➥UnsupportedPhysicalMemoryException

| | |
|---|---|
| ❇ | **UnsupportedPhysicalMemoryException()** |
| ❇ | **UnsupportedPhysicalMemoryException(String descrip tion)** |

## VTMemory — javax.realtime

Object
➥MemoryArea
➥ScopedMemory
➥VTMemory

| | | |
|---|---|---|
| | long | **getMaximumSize()** |
| | String | **toString()** |
| ❇ | | **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 — javax.realtime

Object
➥MemoryArea
➥ScopedMemory
➥VTPhysicalMemory

| | | |
|---|---|---|
| | String | **toString()** |
| ❇ | | **VTPhysicalMemory(Object type, long size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **VTPhysicalMemory(Object type, long base, long size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **VTPhysicalMemory(Object type, long base, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **VTPhysicalMemory(Object type, long size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |

| | | |
|---|---|---|
| ❇ | | **VTPhysicalMemory(Object type, long base, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **VTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException, MemoryInUseException** |
| ❇ | | **VTPhysicalMemory(Object type, SizeEstimator size) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |
| ❇ | | **VTPhysicalMemory(Object type, SizeEstimator size, Runnable logic) *throws* SecurityException, SizeOutOfBoundsException, UnsupportedPhysicalMemoryException, MemoryTypeConflictException** |

## WaitFreeDequeue — javax.realtime

Object
➥WaitFreeDequeue

| | | |
|---|---|---|
| | Object | **blockingRead()** |
| | boolean | **blockingWrite(Object object) *throws* MemoryScopeException** |
| | boolean | **force(Object object)** |
| | Object | **nonBlockingRead()** |
| | boolean | **nonBlockingWrite(Object object) *throws* MemoryScopeException** |
| ❇ | | **WaitFreeDequeue(Thread writer, Thread reader, int maximum, MemoryArea area) *throws* IllegalArgumentException, IllegalAccessException, ClassNotFoundException, InstantiationException** |

## WaitFreeReadQueue — javax.realtime

Object
➥WaitFreeReadQueue

| | | |
|---|---|---|
| | void | **clear()** |
| | boolean | **isEmpty()** |
| | boolean | **isFull()** |
| | Object | **read()** |

| | | |
|---|---|---|
| | int | **size()** |
| | void | **waitForData()** |
| ❇ | | **WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory)** ***throws*** **IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException** |
| ❇ | | **WaitFreeReadQueue(Thread writer, Thread reader, int maximum, MemoryArea memory, boolean notify)** ***throws*** **IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException** |
| | boolean | **write(Object object)** ***throws*** **MemoryScopeException** |

## WaitFreeWriteQueue javax.realtime

Object
➥WaitFreeWriteQueue

| | | |
|---|---|---|
| | void | **clear()** |
| | boolean | **force(Object object)** ***throws*** **MemoryScopeException** |
| | boolean | **isEmpty()** |
| | boolean | **isFull()** |
| | Object | **read()** |
| | int | **size()** |
| ❇ | | **WaitFreeWriteQueue(Thread writer, Thread reader, int maximum, MemoryArea memory)** ***throws*** **IllegalArgumentException, InstantiationException, ClassNotFoundException, IllegalAccessException** |
| | boolean | **write(Object object)** ***throws*** **MemoryScopeException** |

# Bibliography


1. J.H. Anderson, S. Ramamurthy, and K. Jeffay, *Real-Time Computing with Lock-Free Shared Objects*, IEEE Real-Time Systems Symposium 1995, pp. 28-37.
2. J. Anderson, R. Jain, S. Ramamurthy, *Wait-Free Object-Sharing Schemes for Real-Time Uniprocessors and Multiprocessors*, IEEE Real-Time Systems Symposium 1997, pp. 111-122.
3. H. Attiya and N.A. Lynch, *Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty*, IEEE Real-Time Systems Symposium 1989, pp. 268-284.
4. T.P. Baker and A. Shaw, *The Cyclic Executive Model and Ada*, IEEE Real-Time Systems Symposium 1988, pp. 120-129.
5. T.P. Baker, *A Stack-Based Resource Allocation Policy for Realtime Processes*, IEEE Real-Time Systems Symposium 1990, pp. 191-200.
6. T. Baker and O. Pazy, *Real-Time Features for Ada 9X*, IEEE Real-Time Systems Symposium 1991, pp. 172-180.
7. S.K. Baruah, A.K. Mok, and L.E. Rosier, *Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor*, IEEE Real-Time Systems Symposium 1990, pp. 182-190.
8. L. Carnahan and M. Ruark (eds.), *Requirements for Real-Time Extensions for the Java Platform*, National Institute of Standards and Technology, September 1999. Available at http://www.nist.gov/rt-java.
9. P. Chan, R. Lee, and D. Kramer, *The Java Class Libraries, Second Edition, Volume 1, Supplement for the Java 2 Platform, Standard Edition, v1.2*, Addison-Wesley, 1999.
10. M.-Z. Chen and K.J. Lin, *A Priority Ceiling Protocol for Multiple-Instance Resources*, IEEE Real-Time Systems Symposium 1991, pp. 140-149.
11. S. Cheng, J.A. Stankovic, and K. Ramamritham, *Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems*, IEEE Real-Time Systems Symposium 1986, pp. 166-174.
12. R.I. Davis, K. W. Tindell, and A. Burns, *Scheduling Slack Time in Fixed Priority Preemptive Systems*, IEEE Real-Time Systems Symposium 1993, pp. 222-231.
13. B.O. Gallmeister and C. Lanier, *Early Experience with POSIX 1003.4 and POSIX 1003.4a*, IEEE Real-Time Systems Symposium 1991, pp. 190-198.
14. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-

Wesley, 1996.

15. M.L. Green, E.Y.S. Lee, S. Majumdar, D.C. Shannon, *A Distributed Real Time Operating System*, IEEE Real-Time Systems Symposium 1980, pp. 175-184.
16. M.G. Harbour, M.H. Klein, and J.P. Lehoczky, *Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority*, IEEE Real-Time Systems Symposium 1991, pp. 116-128.
17. F. Jahanian and A.K. Mok, *A Graph-Theoretic Approach for Timing Analysis in Real Time Logic*, IEEE Real-Time Systems Symposium 1986, pp. 98-108.
18. K. Jeffay, *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints*, IEEE Real-Time Systems Symposium 1989, pp. 295-307.
19. K. Jeffay, D.F. Stanat, and C.U. Martel , *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks*, IEEE Real-Time Systems Symposium 1991, pp. 129-139.
20. K. Jeffay, *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems*, IEEE Real-Time Systems Symposium 1992, pp. 89-99.
21. K. Jeffay and D.L. Stone, *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems*, IEEE Real-Time Systems Symposium 1993, pp. 212-221.
22. K. Jeffay and D. Bennett, *A Rate-Based Execution Abstraction for Multimedia Computing*, Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (Apr. 1995).
23. E.D. Jensen, C.D. Locke, and H. Tokuda, *A Time-Driven Scheduling Model for Real-Time Operating Systems*, IEEE Real-Time Systems Symposium 1985, pp. 112-133.
24. Mark S. Johnstone, *Non-Compacting Memory Allocation and Real-Time Garbage Collection*, Ph.D. dissertation, The University of Texas at Austin, December 1997.
25. M.B. Jones, *Adaptive Real-Time Resource Management Supporting Modular Composition of Digital Multimedia Services*, Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (Nov. 1993).
26. M.B. Jones, P.J. Leach, R.P. Draves, and J.S. Barrera, *Support for User-centric Modular Real-Time Resource Management in the Rialto Operating System*, Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (Apr. 1995).
27. I. Lee and S.B. Davidson, *Protocols for Timed Synchronous Process Communications*, IEEE Real-Time Systems Symposium 1986, pp. 120-137.

28. J.P. Lehoczky, L. Sha, and J.K. Strosnider, *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*, IEEE Real-Time Systems Symposium 1987, pp. 261-270.

29. J. Lehoczky, L. Sha, and Y. Ding, *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*, IEEE Real-Time Systems Symposium 1989, pp. 166-171.

30. J.P. Lehoczky and T.P. Baker, *Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines*, IEEE Real-Time Systems Symposium 1990, pp. 201-213.

31. J.P. Lehoczky and S. Ramos-Thuel, *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive System*, IEEE Real-Time Systems Symposium 1992, pp. 110-124.

32. K.-J. Lin, S. Natarajan, and J.W.-S. Liu, *Imprecise Results: Utilizing Partial Computations in Real-Time Systems*, IEEE Real-Time Systems Symposium 1987, pp. 210-218.

33. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley, 1999.

34. C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, JACM 20, 1 (Jan. 1973), pp. 46-61.

35. J.W.-S. Liu, K.-J. Lin, and S. Natarajan, *Scheduling Real-Time, Periodic Jobs Using Imprecise Results*, IEEE Real-Time Systems Symposium 1987, pp. 252-260.

36. C. Lizzi, *Enabling Deadline Scheduling for Java Real-Time Computing*, IEEE Real-Time Systems Symposium 1999.

37. C.D. Locke, D.R. Vogel, and T.J. Mesler, *Building a Predictable Avionics Platform in Ada: A Case Study*, IEEE Real-Time Systems Symposium 1991, pp. 180-189.

38. N. Lynch and N. Shavit, *Timing-Based Mutual Exclusion*, IEEE Real-Time Systems Symposium 1992, pp. 2-11.

39. C.W. Mercer and H. Tokuda, *Preemptibility in Real-Time Operating Systems*, IEEE Real-Time Systems Symposium 1992, pp. 78-88.

40. C.W. Mercer, S. Savage, and H. Tokuda, *Processor Capacity Reserves for Multimedia Operating Systems*, Proceedings of the IEEE International Conference on Multimedia Computing and Systems (May 1994).

41. A. Miyoshi, T. Kitayama, H. Tokuda, *Implementation and Evaluation of Real-Time Java Threads*, IEEE Real-Time Systems Symposium 1997, pp. 166-175.

42. J.S. Ostroff and W.M. Wonham, *Modelling, Specifying and Verifying Real-Time*

*Embedded Computer Systems*, IEEE Real-Time Systems Symposium 1987, pp. 124-132.

43. *Portable Operating System Interface (POSIX®) Part 1: System Application Program Interface*, International Standard ISO/IEC 9945-1: 1996 (E) IEEE Std 1003.1, 1996 Edition, The Institute of Electrical and Electronics Engineers, Inc. 1996.

44. R. Rajkumar, L. Sha, and J.P. Lehoczky, *On Countering the Effects of Cycle-Stealing in a Hard Real-Time Environment*, IEEE Real-Time Systems Symposium 1987, pp. 2-11.

45. R. Rajkumar, L. Sha, and J.P. Lehoczky, *Real-Time Synchronization Protocols for Multiprocessors*, IEEE Real-Time Systems Symposium 1988, pp. 259-271.

46. S. Ramos-Thuel and J.P. Lehoczky, *On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems*, IEEE Real-Time Systems Symposium 1993, pp. 160-171.

47. L. Sha, J.P. Lehoczky, and R. Rajkumar, *Solutions for Some Practical Problems in Prioritized Preemptive Scheduling*, IEEE Real-Time Systems Symposium 1986, pp. 181-193.

48. L. Sha, R. Rajkumar, and J. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, Sept., 1990.

49. L. Sha, R. Rajkumar, and J. Lehoczky, *Real-Time Computing using Futurebus+*, IEEE Micro, June, 1991.

50. A.C. Shaw, *Software Clocks, Concurrent Programming, and Slice-Based Scheduling*, IEEE Real-Time Systems Symposium 1986, pp. 14-19.

51. F. Siebert, *Real-Time Garbage Collection in Multi-Threaded Systems on a Single Processor,* IEEE Real-Time Systems Symposium 1999.

52. B. Sprunt, J. Lehoczky, and L. Sha, *Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm*, IEEE Real-Time Systems Symposium 1988, pp. 251-258.

53. Sun Microsystems, Inc., *The Java Community Process Manual*, December 1998, Available at http://java.sun.com/aboutJava/communityprocess/java_community_process.html.

54. S.R. Thuel and J.P. Lehoczky, *Algorithms for Scheduling Hard Aperiodic Tasks in Fixed-Priority Systems Using Slack Stealing*, IEEE Real-Time Systems Symposium 1994, pp. 22-35.

55. H. Tokuda, J.W. Wendorf, and H.-Y. Wang, *Implementation of a Time-Driven Scheduler for ReaI-Time Operating System*, IEEE Real-Time Systems Sympo-

sium 1987, pp. 271-280.

56. D.M. Washabaugh and D. Kafura, *Incremental Garbage Collection of Concurrent Objects for Real-Time Applications*, IEEE Real-Time Systems Symposium 1990, pp. 21-31.
57. P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review,* In International Workshop on Memory Management, Kinross, Scotland, UK, September 1995.
58. W. Zhao and K. Ramamritham, *A Virtual Time CSMA Protocol for Hard Real Time Communication*, IEEE Real-Time Systems Symposium 1986, pp. 120-127.
59. W. Zhao and J.A. Stankovic, *Performance Analysis of FCFS and Improved FCFS Scheduling Algorithms for Dynamic Real-Time Computer Systems*, IEEE Real-Time Systems Symposium 1989, pp. 156-165.

# Index

# H



# I

## J



## L



## M

# O



# P

# T

## U



## V



## W