AERO: Architecture for Enhanced Reprogrammability and Operability

Contract ESTEC 15750/02/NL/LVH

Specification

(Extract of original document)

F. Deladerrière, Astrium SaS
F.Siebert, Aicas Gmbh
T.Ritzau, Linköping Universitet

Reference: AERO/SP1
Issue: 0.3
Date: **2002-09-09**

Abstract:

This document is the specification of the AERO Real-time Java Virtual Machine project, output of the task 1.a.2.

This document defines functional requirements for AERO JVM, who is a Java Virtual Machine with real-time capacities, for ERC32 processor. This document contains an overview of AERO JVM functionalities, functional requirements, representative's requirements, environment requirement, operability requirements, portability and maintenance requirements.

Written by:	**Name**	**Company**	**Signature**	**Internal reference**
	F. Deladerrière	Astrium		

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (Contract ESTEC 15750/02/NL/LVH) conducted by a consortium led by ASTRIUM-SAS with Aicas Gmbh and Linköping Universitet. For more information please contact:

Frank J. de Bruin
ESTEC. Keplerlaan 1. PO Box 299
2200 AG Noordwijk ZH – The Netherlands
Tel: +31 (0) 71 565 4951. Fax: +31 (0) 71 565 5420
e-mail fdebruin@estec.esa.nl

Frédéric Deladerrière
ASTRIUM
31, avenue des cosmonautes
F-31 402 Toulouse Cedex 4, France
Tel: +33 5 62 19 56 49. Fax: +33 5 62 19 78 97
e-mail: frederic.deladerriere@astrium-space.com

Fridtjof Siebert
AICAS Gmbh
Hoepfner Burg
Haid-und-Neu-StraBe 18
D-76131 KarlsRuhe, Germany
Tel: +49 721 663 96823 Fax: +49 721 663 96893
e-mail: siebert@aicas.com

Tobias Ritzau
Linköping Universitet
Dep. Of Computer and Information Science
SE-58183 Linköping, Sweden
Tel: +46 13 28 4494. Fax: +46 13 28 5899

e-mail: tobri@ida.liu.se

Revision History

Version	Date	Paragraphs modified	Comments
0.1	2002-04-15		First issue
0.2	2002-07-15	2.2, 3.1.1,3.1.2,4.2, 7.1	PDR comments taken into account
0.3	2002-09-09	All documents	Grammar and spelling corrections

Table of Contents

1. Introduction

1.1 Scope

1.1.1 Scope of the Project

AERO (Architecture for Enhanced Reprogrammability and Operability) is an ESA project (contract ESTEC 15750/02/NL/LVH). The objectives of the project are to investigate on a real-time Java virtual machine for ERC32. Special attention is put on the garbage collection mechanism and deterministic execution model.

The project is split in two phases. The first phase investigates existing virtual machine to choose a potential candidate that will be customized, are then investigates the definition of requirements concerning a real-time interpreter in on-board systems. An implementation plan is proposed for the second phase. This second phase is dedicated to the definition of software functions of the real-time Java virtual machine and to their implementation and assessment through validation tests.

1.1.2 Scope of the Document

This document is an output of the task 1.a.2 “Software JVM Specification”.

This document defines functional requirements for AERO JVM, is a Java Virtual Machine with real-time capacities, for ERC32 processor.

On board application programs shall be written in a standard language : Java, which is compiled to give Bytecode. Then this code can be loaded in spacecraft to be executed by the virtual machine that is a part of on board software.

This document contains :

- an overview of AERO JVM functionalities and standard JVM mechanisms
- functional requirements
- environment requirement
- operability requirements
- portability and maintenance requirements.

Important notice :

Due to complexity of JVM concept, the overview chapter introduces with precision the mechanisms of standard JVM to provide easier requirements understanding.

1.2 Related Documentation

[RTSJ] Real-Time Specification for Java (RT for Java Expert Group) final release, December 2001.

[JSL] Java Specification Language, Bill Joy, Guy Steele, James Gosling, Gilad Bracha, 2000 2nd Edition ISBN 0-20131-008-2

[JVMS] Java Virtual Machine Specification, Tim Lindholm & Frank Yellin Addison- Wesley Pub Co, 1999 2nd Edition ISBN 0-20-143294-3

[BOOK1] Inside Java 2 Virtual Machine, B.Veners, Mac Graw Hill, 1999 2nd Edition ISBN 0-07-135093-4

[BOOK2] Java Virtual Machine, Jon Meyer & Troy Downing, O'Reilly, ISBN 1-56592-194-1

1.3 Applicable Documentation

[AERO] Architecture for Enhanced Reprogrammability and Operability, ESTEC Contract n°15750/02/NL/LVH.

[Prop] Architecture for Enhanced Reprogrammability and Operability, Proposal for ESA ITT AO/1-3959/01/NL/PB. Astrium EEA.PR.FD.3682269.01.

[MNM] Minutes of AERO Project Negotiation Meeting, Noordwijk, NL, January 31, 2002

[MP] Management Plan of AERO Project

1.4 Definition of Terms and Acronyms

1.4.1 Definition of Terms

None

1.4.2 Acronyms and Abbreviations

Acronyms and abbreviations used in this text are defined as follows :

AERO Architecture for Enhanced Reprogrammability and Operability

AIE - AsynchronouslyInterruptedException

AI-method - (Asynchronously Interruptible) A method is said to be asynchronously interruptible if it includes AIE in its throws clause.

ATC Asynchronous Transfer of Control

ATC-deferred section - a synchronized method, a synchronized statement, or any method or constructor without AIE in its throws clause.

ESA European Space Agency

ESTEC European Space Technological Centre

GC Garbage Collector

ICD Interface Control Document

ICR Individual Control Register

JNI Java Native Interface

JVM Java Virtual Machine

OBS On Board Software

RTSJ Real-Time Specification for Java

TBC To Be Confirmed

TBD To Be Defined

TN Technical Note

VM Virtual Machine

WP Work Package

2. Overview

CHAPTER REMOVED
FROM THE AERO PROJECT
ORIGINAL SP1 DOCUMENT

3. Functional Requirements

This chapter defines functional requirements for general use of the AERO JVM.

The SRD requirements are introduced trough 5 column tables (to be used in tracability tool):

- the first column identifies the SRD requirement with following rules :
  REQ/AERO. xxx (for function). xxx (for sub-function). xxxx (number)

  Functions identifiers are ENV for environment, GEN for general, DES for design, RT for real-time, TOO for tools

  Sub-function identifiers are INIT for initialisation, DBG for debug, RUN for running, VER for verification, TRA for trace, SCH for scheduling, SYN for synchroning, ASY from asynchroning, THR for thread, MEM for memory

- the second column describes the requirement.
- the third column is empty , link with URD is not applicable for AERO JVM
- the fourth column indicate the level of compliance of the requirement : STD (Standard Java core), RTSJ (Real-Time Specification for Java), Aero (specific to project)
- the fifth column gives the corresponding verification method with :

  T : testing,
  A : analysis,
  I : inspection code.

When a requirement will be suppressed identifier will not be suppressed, deleted mention will replace description

3.1 Java Virtual Machine

3.1.1 Design

General

Design general requirement defines the base capabilities of the AERO-VM, including the compliance with the JVM core standard. One of the most important features of Java is the ability to dynamically load code in the form of class files during execution, be it from local files or from a remote system. Performance is another important requirement, but on board constraints involve to provide other techniques than standard ones (like JIT technologies).

REQ/AERO.DES.GEN.0010	The AERO JVM shall implement base mechanism as defined in [JVMS] and interface of the standard core JVM including Classloader, Garbage Collector, Memory Manager and Security Manager.		STD	T
REQ/AERO.DES.GEN.0020	The AERO JVM shall supports dynamic class loading. *Justification :* Any software component can be loaded dynamically, allowing on-the-fly reconfiguration, hot swapping of code, dynamic additions of new features and application execution.		STD	T
REQ/AERO.DES.GEN.0030	Optimisations shall be developed to ensure good performance of the java code execution *Remark :*verification shall be made using standard Java Benchmark tools		Aero	T
REQ/AERO.DES.GEN.0040	Just-in-time compilation technologies shall not be used. *Remark :* the initial delay for compilation is breaking all real-time constraint.		Aero	I

REQ/AERO.DES.GEN.0050	The AERO JVM shall execute up to 64 tasks without requiring to run a new instance of the JVM.		Aero	T
REQ/AERO.DES.GEN.0060	The Start process of an application shall be implemented as: loading application, create new instance of corresponding object, start the associated process, and run the application		Aero	T

Scheduling

In the AERO JVM, scheduling refers to the production of a sequence (or ordering) for the execution of a set of threads (*a schedule*).

Requirements on scheduling have a direct impact on the design of the solution.

REQ/AERO.DES.SCH.0010	Execution of machine instructions shall be predictable and in conformance with [RTSJ]		Aero	I
REQ/AERO.DES.SCH.0020	A real-time Scheduler shall be provided instead of standard Java scheduler, expected WCET to react on an event shall be fewer than 10 milliseconds.		Aero	T
REQ/AERO.DES.SCH.0030	A generic (java standard interface) Schedulable interface will be provided *Justification :*this interface will be used to specify that an object is schedulable by the real-time scheduler		RTSJ	T
REQ/AERO.DES.SCH.0040	Any instance of any class implementing Schedulable shall be a schedulable object.		RTSJ	T
REQ/AERO.DES.SCH.0050	Schedulable objects scheduling and dispatching shall be managed by the instance of Scheduler		RTSJ	T
REQ/AERO.DES.SCH.0060	In conformance with RTSJ, three classes (and corresponding JVM internal execution model) shall be implemented in AERO JVM : RealtimeThread, NoHeapRealtimeThread and AsyncEventHandler.		RTSJ	T

Memory

Real-time constraints introduce the concept of memory area. A memory area represents an area of memory that may be used for the allocation of objects. Some memory areas exist outside 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 any object within the heap to preserve the integrity of the heap.

REQ/AERO.DES.MEM.0010	To be compliant with [RTSJ] four types of memory areas shall be provided : scoped, physical, immortal and heap *Justification :* - Scoped memory provides a mechanism for dealing with a class of objects that have lifetime defined by syntactic scope - Physical memory allows java objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access - 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 - Heap memory represents an area of memory that is the heap. The determinant of lifetime of objects on the heap is unchanged (compare with standard Java implementation), the lifetime is still determined by visibility.		RTSJ	T
REQ/AERO.DES.MEM.0020	AERO-VM shall provide support for memory allocation budgets for threads using memory areas. Maximum memory area consumption and maximum allocation rates for individual real-time threads shall be specified by the user when the thread is created		Aero	A

REQ/AERO.DES.MEM.0030	AERO-VM shall provide deterministic real-time garbage collection based on a known and proven technique.		Aero	A
REQ/AERO.DES.MEM.0040	AERO-VM garbage collection shall be exact (in contrast to being conservative), i.e. all unreachable objects must be reclaimed.		Aero	A
REQ/AERO.DES.MEM.0050	It must be possible to predict the amount of garbage collection work needed by any piece of code, i.e. it must be possible to calculate the WCET of any piece of code regardless if contains memory management or not.		Aero	A
REQ/AERO.DES.MEM.0060	Given a maximum memory usage and the amount of available memory, it must be possible to prove that the system never runs out of memory.		Aero	A

Thread

REQ/AERO.DES.THR.0010	To provide robust execution model and performance, all java threads provided by the AERO-VM shall be real-time threads.		Aero	T
REQ/AERO.DES.THR.0020	Threads waiting to acquire a resource shall be released in execution eligibility order based on their priorities. *Remark :* Calling a thread with lower priority means to increase priority of this thread temporarily to not be stopped during the call.		RTSJ	T
REQ/AERO.DES.THR.0030	Threads waiting to enter synchronized blocks shall be granted access to the synchronized block in execution eligibility order		RTSJ	T
REQ/AERO.DES.THR.0040	A blocked thread that becomes ready to run shall be given access to the processor in execution eligibility order		RTSJ	T
REQ/AERO.DES.THR.0050	A thread that performs a ‘yield’ shall be given access to the processor after waiting threads of the same execution eligibility		RTSJ	T

Asynchronous Event handling

Real-time systems typically interact closely with the real world. With respect to the execution of logic, the real world is asynchronous. We thus felt compelled to include efficient mechanisms for programming disciplines that would accommodate this inherent asynchrony. The real-time specification 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.

REQ/AERO.DES.ASY.0010	Asynchronous event facility shall be provided. In conformance with [RTSJ] two classes shall be available : AsyncEvent and AsyncEventHandler		RTSJ	T
REQ/AERO.DES.ASY.0020	AsyncEvent object shall manage the unblocking of handlers when event is fired, and the set of handlers associated with the event (cf. corresponding API)		RTSJ	T
REQ/AERO.DES.ASY.0030	AsyncEventHandler object shall be a java runnable event handler object with parameters to control execution of handler once the associated AsyncEvent is fired.		RTSJ	T
REQ/AERO.DES.ASY.0040	When an event is fired, the handler shall be executed asynchronously, scheduled according to the associated parameters (cf. corresponding API)		RTSJ	T
REQ/AERO.DES.ASY.0050	The system must cope well with situations where there are 100 instances of AsyncEvent and AsyncEventHandler. The number of fired (in process) handlers is expected to be smaller.		RTSJ	T
REQ/AERO.DES.ASY.0060	New Timer class shall be a specialized form of an AsyncEventHandler that represents an event whose occurrence is driven by time.		RTSJ	T
REQ/AERO.DES.ASY.0070	There must be two forms of Timers to be compliant with [RTSJ]: the OneShotTimer and the PeriodicTime. Instance of OneShotTimer fire once, at the specified time. Periodic timers fire off at the specified time, and then periodically according to a specified interval.		RTSJ	T
REQ/AERO.DES.ASY.0080	A specific object must drive timers: Clock that represents the real-time clock. The Clock class		RTSJ	T

	may be extended to represent other clocks.			
REQ/AERO.DES.ASY.0090	In conformance with [RTSJ], the clock class provide the getRealtimeClock() method.		RTSJ	T

Asynchronous Transfer of Control(ATC)

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. A mechanism which extends Java's exception handling shall be include to allow applications to programmatically change the locus of control of another Java thread. It is important to note that in the RTSJ this asynchronous transfer of control is restricted to logic specifically written with the assumption that its locus of control may asynchronously change.

REQ/AERO.DES.ASY.0100	A mechanism shall be providing 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)		RTSJ	T
REQ/AERO.DES.ASY.0110	A thread shall explicitly indicate its susceptibility to ATC. *Remark:* 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).		RTSJ	T
REQ/AERO.DES.ASY.0120	Even if a thread allows ATC, some code sections shall be executed to completion and thus ATC is deferred in such sections. *Justification: The* ATC-deferred sections are synchronized methods and statements.		RTSJ	T
REQ/AERO.DES.ASY.0130	Code that responds to an ATC shall not return to the point in the thread where the ATC was triggered; that is, an ATC is an unconditional transfer of control. Presumptive 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).		RTSJ	T
REQ/AERO.DES.ASY.0140	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.		RTSJ	T

REQ/AERO.DES.ASY.0150	Through ATC it shall be possible to abort a thread but in another manner that does not carry the dangers of the Thread class's stop() and destroy() methods.		RTSJ	I
REQ/AERO.DES.ASY.0160	If ATC is modeled by exception handling, there shall 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		RTSJ	I
REQ/AERO.DES.ASY.0170	Nested ATCs must work in conformance with RSTJ. *Remarks :* 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 ATCdeferred 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.		RTSJ	T
REQ/AERO.DES.ASY.0190	ATC must be implemented without inducing an overhead for programs that do not use it		RTSJ	A
REQ/AERO.DES.ASY.0200	If code with a timeout completes before the timeout's deadline, the timeout shall be automatically stopped and corresponding resources returned to the system		RTSJ	T

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, as defined in [RTSJ] mechanism for asynchronous event handling and transfer of control is safe.

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. A goal 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 corresponding algorithm can consist of a number of asynchronous event handlers that are bound to external events.

REQ/AERO.DES.ASY.0210	When an asynchronous event occurs the handlers shall invoke interrupt() on appropriate threads.		RTSJ	T
REQ/AERO.DES.ASY.0220	Threads that are terminated 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. *Remark* :This idiom provides a quick (if coded to be so) but orderly clean up and termination of the thread.		RTSJ	T
REQ/AERO.DES.ASY.0230	The system shall comprise 10 asynchronous event handlers as appropriate. This number could be change at AERO-VM generation.		Aero	T

Exceptions & Errors

Real-time problematic require to introduce several new exceptions, and some new treatment of exceptions surrounding asynchronous transfer of control and memory allocators.

REQ/AERO.DES.EXE.0010	In conformance with RTSJ, new exceptions compare with standard Java and associated mechanisms shall be implanted: - *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 Release Parameter 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.		RTSJ	T

REQ/AERO.DES.EXE.0020	New runtime exceptions and associated mechanisms shall be implanted : - *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.		RTSJ	T
REQ/AERO.DES.EXE.0030	New error and associated mechanisms shall be implanted : - *ResourceLimitError:* Thrown if an attempt is made to exceed a system resource limit, such as the maximum number of locks.		RTSJ	T

JVM conformance table

Function	**Standard JVM core**	**Required in AERO-VM**	**Remarks**
General			
Dynamic class loading	Yes	Yes	
Garbage collector	Yes	Yes	Specific GC for real-time
Memory manager	Yes	Yes	
Security manager	Yes	Yes	
Just-in-Time compiler	No	No	
Single JVM	No	Yes	Use a single instance to run any java applications
Scheduling			
Predictable execution	No	Yes	
Real-time scheduler	No	Yes	
Schedulable interface	Yes	Yes	
Specific thread	No	Yes	RealtimeThread, NoHeapRealtimeThread
Asynchronous handler	No	Yes	
Memory			
Memory types	No	Yes	
Memory allocation for thread	No	Yes	
Real-time GC	No	Yes	

Function	Standard JVM core	AERO-VM	Remarks
Thread			
Classical thread	Yes	Yes	Classical thread as assimilated to real-time one in AERO-VM
Real-time thread	No	Yes	
Asynchronous			
Asynchronous event class	No	Yes	
Asynchronous handler	No	Yes	
Timer	Yes	Yes	Standard Timer class is not a real-time implementation
PeriodicTimer	No	Yes	
ATC	No	Yes	
Asynchronous thread term.	Partially	Yes	Standard thread termination is not deterministic
Exception & Errors			
Standard mechanism	Yes	Yes	
Standard class	Yes	Yes	
Asynchronous exceptions	No	Yes	
Asynchronous error	No	Yes	

3.1.2 Real-time

Init

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.

REQ/AERO.RT.INI.0010	The POSIX signal handler class shall be available (AERO JVM executes on an underlying platform that provides a subset of signals named with the POSIX names).		Aero	T
REQ/AERO.RT.INI.0020	The RealtimeSecurity class is required as defined in [RTSJ]		RTSJ	T

Thread

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 RealtimeParameters and Memory Parameters provided to the RealtimeThread constructor allow for 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.

REQ/AERO.RT.THR.0010	Specific classes shall be provided to allow creation of threads that have more precise scheduling semantics than java.lang.thread		RTSJ	T
REQ/AERO.RT.THR.0020	This classes shall allow the use of areas of memory other than the heap for the allocation of objects. They must allow the definition of methods for handling asynchronously interrupted and provide the scheduling semantics for handling asynchronous events.		RTSJ	T
REQ/AERO.RT.THR.0030	The default scheduling policy shall manage the execution of instances of Object that implement the interface Schedulable.		RTSJ	T
REQ/AERO.RT.THR.0040	Any scheduling policy presents in an implementation shall be available to instances of objects which implement the interface Schedulable		RTSJ	T
REQ/AERO.RT.THR.0050	The function of allocating objects in memory areas defined by instances of ScopedMemory or its subclasses shall be available only to logic within instances of RealtimeThread, NoHeapRealtimeThread, AsyncEventHandler and BoundAsyncEventHandler.		RTSJ	T
REQ/AERO.RT.THR.0060	The invocation of methods that throw AsynchronouslyInterruptedException shall have effect only when the invocation occurs in the context of instances of RealtimeThread, NoHeapRealtimeThread, AsyncEventHandler and BoundAsyncEventHandler.		RTSJ	T

	Remark: Chosen AERO-VM implementation is fully compliant of RTSJ requirement.			
REQ/AERO.RT.THR.0070	In the specific case in which an instance of NoHeapRealtimeThread and instance of either RealtimeThread or Thread synchronize on the same object an exception to the immediately previous statement applies. This exception has the effect of causing an instance of NoHeapRealtimeThread to wait for the garbage collector; exception is transferred to the immediate previous bytecode instruction, that produce a bytecode exception.		RTSJ	T
REQ/AERO.RT.THR.0080	If GC implementation is made at thread level, RealtimeThread class instance shall have an execution eligibility lower than garbage collector. *Remark* : GC implementation may be not at thread level		RTSJ	I
REQ/AERO.RT.THR.0090	Changing values in SchedulingParameters, ProcessingParameters, ReleaseParameters, ProcessingGroupParameters, or use of Thread.setPriority() must not affect the correctness of any implemented priority inversion avoidance algorithm.		RTSJ	T
REQ/AERO.RT.THR.0100	Instances of objects which implement the interface Schedulable shall inherit the scope stack of the thread invoking the constructor. *Justification :* 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 of context of the thread invoking the constructor (RTJS requirement).		RTSJ	I
REQ/AERO.RT.THR.0110	Instances of objects which implement the interface Schedulable shall 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.		RTSJ	I
REQ/AERO.RT.THR.0120	The default parameter values for an object implementing the interface Schedulable must be the parameter values of the thread invoking the constructor.		RTSJ	I

	Justification : If the thread invoking the constructor does not have parameter values then the default values are those values associated with the instance if Scheduler which will manage the object.			
REQ/AERO.RT.THR.0130	Instance of objects implementing the interface Schedulable shall be placed in memory represented by instances of ImmortalMemory, HeapMemory, LTPhysicalMemory, VTPhysicalMemory, or ImmortalPhysicalMemory.		RTSJ	I

Scheduling

As specified the required semantics and requirements of this section establish a scheduling policy that is very similar to the scheduling policies found on the vast majority of real-time operating systems and kernels in commercial use today. The 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). [RTSJ] 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.

REQ/AERO.RT.SCH.0010	The base scheduler shall support at least 28 unique values in the priorityLevel field of an instance of PriorityParameters (RTJS minimum compliance requirement) *Justification* : current onboard interpreter use 3 priorities		RTSJ	T
REQ/AERO.RT.SCH.0020	Higher values in the priorityLevel field of an instance of PriorityParameters must have a higher execution eligibility		RTSJ	T
REQ/AERO.RT.SCH.0030	In unique means that if two schedulable objects have different values in the priorityLevel field in their respective instance of PriorityParameters, the schedulable object with the higher value shall always execute in preference to the schedulable object with the lower value when both are ready to execute.		RTSJ	T
REQ/AERO.RT.SCH.0040	Native priorities which are lower than the 28 required real-time priorities shall be available. These are to be used for regular Java threads (ie instance of threads which are not instances of RealtimeThread, NoHeapRealtimeThread or AsyncEventHandler classes or subclasses). The ten traditional Java thread priorities shall have an arbitrary mapping into the native priorities. These ten traditional Java thread priorities and the required minimum 28 unique real-time thread		RTSJ	T

	priorities shall be from the same space. Assignment of any of this (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 (RTJS requirement).			
REQ/AERO.RT.SCH.0050	The dispatching mechanism must allow the pre-emption of the execution of schedulable objects at a point not governed by the pre-empted object.		RTSJ	A
REQ/AERO.RT.SCH.0060	For schedulable objects managed by the base scheduler no part of the system shall 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.		RTSJ	T
REQ/AERO.RT.SCH.0070	All instances of RelativeTime used in instances of ProcessingParameters, Scheduling Parameters, and ReleaseParameters shall be measured from the time at which the associated thread (or first such thread) is started.		RTSJ	T
REQ/AERO.RT.SCH.0080	PriorityScheduler.getNormPriority() shall be set to ((Priority- Scheduler.getMaxPriority() - PriorityScheduler.getMinPriority())/3) + PriorityScheduler.getMinPriority().		RTSJ	I
REQ/AERO.RT.SCH.0090	If instances of RealtimeThread or NoHeapRealtimeThread are constructed without a reference to a SchedulingParameters object a SchedulingParameters object must be 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.		RTSJ	T
REQ/AERO.RT.SCH.0100	Feasibility algorithm is not required, the function shall return success whenever the feasibility algorithm is executed *Justification* : the [RTSJ] does not require any particular feasibility algorithm be implemented in the Scheduler object.		Aero	I

REQ/AERO.RT.SCH.0110	For instances of AsyncEventHandler with a release parameters object of type Sporadic Parameters implementations are required to maintain a list of times at which instances of AsyncEvent occurred. The ith time may be removed from the queue after the ith execution of the handleAsyncEvent method.		RTSJ	T
REQ/AERO.RT.SCH.0120	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 shall be handled normally.		RTSJ	T
REQ/AERO.RT.SCH.0130	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 must have precedence over ResourceLimitExceeded.		RTSJ	T
REQ/AERO.RT.SCH.0140	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.		RTSJ	T

Memory

Languages that employ automatic reclamation of blocks of memory allocated in what is traditionally called the heap by program logic also typically use an algorithm called a garbage collector. Garbage collection algorithms and implementations vary in the amount of non-determinacy they add to the execution of program logic. To date, experts 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 MEMORYAREA 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.

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 possible. Thus a thread attempting to enter a scope can only do so by entering in the established nesting order.

REQ/AERO.RT.MEM.0010	Some MemoryArea classes are required to have linear (in object size) allocation time. *Justification :* The linear time attribute requires that, ignoring performance variations due to hardware caches or similar optimizations and execution of any static initialises, 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 constant C.		RTSJ	I
REQ/AERO.RT.MEM.0020	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. *Remark :* The algorithms for maintaining the scope structure are given in "Maintaining the Scope Stack." Of the RTSJ *Justification :*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 Scoped-Memory, all subsequent uses of the new operation will allocate the memory from the area of memory associated with the enclosing scope.		RTSJ	I
REQ/AERO.RT.MEM.0030	The parent of a scoped memory area must be the memory area in which the object representing the scoped memory area is allocated.		RTSJ	I
REQ/AERO.RT.MEM.0040	The *single parent rule* requires that a scope memory area must have exactly zero or one parent.		RTSJ	T
REQ/AERO.RT.MEM.0050	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.*		RTSJ	T
REQ/AERO.RT.MEM.0060	Each instance of the class ScopedMemory or its subclasses must maintain a reference count of		RTSJ	I

	the number of threads in which it is being used. *Remark :* 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.			
REQ/AERO.RT.MEM.0070	Objects created in any immortal memory area shall live for the duration of the application. Their finalizers are only run when the application is terminated.		RTSJ	I
REQ/AERO.RT.MEM.0080	The addresses of objects in any MemoryArea that is associated with a NoHeap- RealtimeThread must remain fixed while they are alive.		RTSJ	I
REQ/AERO.RT.MEM.0090	Each instance of the virtual machine must have exactly one instance of the class ImmortalMemory		RTSJ	I
REQ/AERO.RT.MEM.0100	Each instance of the virtual machine must have exactly one instance of the class HeapMemory.		RTSJ	I
REQ/AERO.RT.MEM.0110	Each instance of the virtual machine shall behave as if there is an area of memory into which all Class objects are placed and which is unexceptionally referenceable by NoHeapRealtimeThreads.		RTSJ	I

<table>
<tr>
<td>REQ/AERO.RT.MEM.0120</td>
<td>Strict assignment rules placed on assignments to or from memory areas must prevent the creation of dangling pointers, and thus maintain the pointer safety of Java. The restrictions are listed in the following table:
<table>
<tr><th></th><th>Reference to Heap</th><th>Ref. To Immortal</th><th>Ref. To Scoped</th></tr>
<tr><td>Heap</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>Immortal</td><td>Yes</td><td>Yes</td><td>No</td></tr>
<tr><td>Scoped</td><td>Yes</td><td>Yes</td><td>Yes if same,outer, shared scope</td></tr>
<tr><td>Local Variable</td><td>Yes</td><td>Yes</td><td>Yes if same,outer, shared scope</td></tr>
</table>
</td>
<td></td>
<td>RTSJ</td>
<td>A</td>
</tr>
<tr>
<td>REQ/AERO.RT.MEM.0130</td>
<td>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).</td>
<td></td>
<td>RTSJ</td>
<td>A</td>
</tr>
</table>

Synchronization

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 [RTSJ] which can enforce mutual exclusion in the traditional sense is the keyword synchronized. Noting that this 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.

REQ/AERO.RT.SYN.0010	Threads waiting to enter synchronized blocks must be priority queue ordered. If threads with the same priority are possible under the active scheduling policy such threads shall be queued in FIFO order.		RTSJ	T
REQ/AERO.RT.SYN.0020	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.		RTSJ	T
REQ/AERO.RT.SYN.0030	The Priority Inheritance monitor control policy must be implemented.		RTSJ	T

Time

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 milliseconds 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 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.

REQ/AERO.RT.GEN.0010	All time objects must maintain microsecond precision and report their values in terms of millisecond and microsecond constituents.		Aero	T
REQ/AERO.RT.GEN.0020	Time objects must be constructed from other time objects, or from millisecond/microseconds values.		Aero	T
REQ/AERO.RT.GEN.0030	Time objects must provide simple addition and subtraction operations, both for the entire object and for constituent parts.		RTSJ	T
REQ/AERO.RT.GEN.0040	Time objects must implement the Comparable interface if it is available. The compareTo() method must be implemented even if the interface is not available.		RTSJ	T
REQ/AERO.RT.GEN.0050	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. *Remark :* 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.		RTSJ	A

Timer
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.

REQ/AERO.RT.GEN.0060	The Clock class shall be capable of reporting the achievable resolution of timers based on that clock.		RTSJ	T
REQ/AERO.RT.GEN.0070	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.		RTSJ	T
REQ/AERO.RT.GEN.0080	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.		RTSJ	T
REQ/AERO.RT.GEN.0090	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.		RTSJ	T

Asynchrony

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 (Asynchronous Transfer of Control) 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 solution 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().

The following terms and abbreviations will be used:

ATC - Asynchronous Transfer of Control
AIE - Asynchronously Interrupted Exception (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.

REQ/AERO.RT.ASY.0010	The Java Real Time approach to ATC shall be designed to be based on exceptions and it shall be an extension of the current Java language rules for java.lang.Thread.interrupt().		RTSJ	T
REQ/AERO.RT.ASY.0020	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() must be scheduled for execution. This action may or may not be idempotent.		RTSJ	T
REQ/AERO.RT.ASY.0030	Every occurrence of an event shall increment a counter in each associated handler.		RTSJ	T
REQ/AERO.RT.ASY.0040	Handlers shall elect to execute logic for each occurrence of the event or not.		RTSJ	T
REQ/AERO.RT.ASY.0050	Instances of AsyncEvent and AsyncEventHandler must be created and used by any program logic.		RTSJ	T
REQ/AERO.RT.ASY.0060	More than one instance of AsyncEventHandler must be added to an instance of AsyncEvent.		RTSJ	T
REQ/AERO.RT.ASY.0070	An instance of AsyncEventHandler must be added to more than one instance of AsyncEvent.		RTSJ	T
REQ/AERO.RT.ASY.0080	Instances of the class AsynchronouslyInterruptedException shall be generated by execution of program logic and by internal virtual machine mechanisms that are asynchronous to the		RTSJ	I

	execution of program logic which is the target of the exception.			
REQ/AERO.RT.ASY.0090	Program logic that exists in methods that throw AsynchronouslyInterrupted-Exception must be subject to receiving an instance of AsynchronouslyInterrupted-Exception at any time during execution except as provided below.		RTSJ	I
REQ/AERO.RT.ASY.0100	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. *Justification :* The implementation, when either AIE.fire() or Realtime-Thread.interrupt() shall be 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.		RTSJ	T
REQ/AERO.RT.ASY.0110	Program logic executing within a synchronized block within a method with AsynchronouslyInterruptedException in its throws clause must not be subject to receiving an instance of AIE. *Justification :* 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.		RTSJ	T
REQ/AERO.RT.ASY.0120	Constructors must be allowed to include AsynchronouslyInterruptedException in their throws clause and will thus be interruptible.		RTSJ	T
REQ/AERO.RT.ASY.0130	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.		RTSJ	A

REQ/AERO.RT.ASY.0140	ATC must work 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.		RTSJ	T
REQ/AERO.RT.ASY.0150	If an AIE is in a pending state then this AIE shall be thrown only when: 1. Control enters an AI-method. 2. Control returns to an AI-method. 2. Control leaves a synchronized block within an AI-method.		RTSJ	T
REQ/AERO.RT.ASY.0160	When inherited (event class) happened() method is called on an AIE or that AIE is superseded by another the first AIE's state must be made non-pending.		RTSJ	T
REQ/AERO.RT.ASY.0170	If the current AIE is an AIE0 and the new AIE is an AIEx associated with any frame on the stack then the new AIE (AIEx) shall be discarded.		RTSJ	T
REQ/AERO.RT.ASY.0180	If the current AIE is an AIEx and the new AIE is an AIE0, then the current AIE (AIEx) shall be replaced by the new AIE (AIE0).		RTSJ	T

REQ/AERO.RT.ASY.0190	If the current AIE is an AIEx and the new AIE is an AIEy from a frame lower on the stack, then the new AIE (AIEy) must be discarded.		RTSJ	T
REQ/AERO.RT.ASY.0200	If the current AIE is an AIEx and the new AIE is an AIEy from a frame higher on the stack, the current AIE (AIEx) must be replaced by the new AIE (AIEy).		RTSJ	T

Remark :

An AIE may be raised while another AIE is pending or in action. Because AI code blocks are nested by method invocation (a stack-based nesting) there is a natural SEMANTICS AND REQUIREMENTS precedence among active instances of AIE. Let AIE0 be the AIE raised when t.interrupt() is invoked and AIEi (i = 1,...,n, for n unique instances of AIE) be the AIE raised when AIEi.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.

Match	No Match
Propagate == true clear the pending AIE, return true Propagate == false clear the pending AIE, return false	propagate (whether the AIE remains pending is invisible except to the implementation) do not clear the pending AIE, return false

Exception

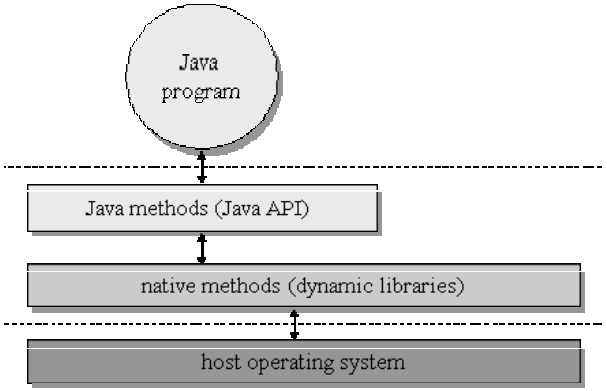
The need for additional exceptions given the new semantics added by the other sections of this specification is obvious. That the specification attaches new, nontraditional, exception semantics to AsynchronouslyInterruptedException is, perhaps, not so obvious.

REQ/AERO.RT.EXC.0010	All exceptions, except AsynchronouslyInterruptedException, are required to have semantics exactly as those of their eventual superclass in the java.* hierarchy.		RTSJ	T
REQ/AERO.RT.EXC.0020	Instances of the class AsynchronouslyInterruptedException shall 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.		RTSJ	T
REQ/AERO.RT.EXC.0030	Program logic that exists in methods that throw AsynchronouslyInterrupted-Exception shall be subject to receiving an instance of AsynchronouslyInterrupted-Exception at any time during execution.		RTSJ	T

3.2 API

The Java API helps make Java suitable for networks through its support for platform independence and security. The Java API is set of runtime libraries that give a standard way to access the system resources of a host computer. When writing a Java program, the base mechanism assume the class files of the Java API will be available at any Java virtual machine that may ever have the privilege of running the program. This is a relatively safe assumption because the Java virtual machine and the class files for the Java API are the required components of any implementation of the Java Platform. When running a Java program, the virtual machine loads the Java API class files that are referred to by the program's class files. The combination of all loaded class files (from the program and from the Java API) and any loaded dynamic libraries (containing native methods) constitute the full program executed by the Java virtual machine.

The class files of the Java API are inherently specific to the host platform. The API's functionality must be implemented expressly for a particular platform before that platform can host Java programs. To access the native resources of the host, the Java API calls native methods. As shown in next figure, the class files of the Java API invoke native methods so the Java program doesn't have to. In this manner, the Java API's class files provide a Java program with a standard, platform-independent interface to the underlying host. To the Java program, the Java API looks the same and behaves predictably no matter what platform happens to be underneath. Precisely because the Java virtual machine and Java API are implemented specifically for each particular host platform, Java programs themselves can be platform independent.



The internal design of the Java API is also geared towards platform independence. With the aim of making the execution its best on each platform, the virtual machine will very likely adapt elements of application slightly differently on different platforms. In these ways and many others, the internal architecture of the Java API is aimed at facilitating the platform independence of the Java programs that use it.

In addition to facilitating platform independence, the Java API contributes to Java's security model. The methods of the Java API, before they perform any action that could potentially be harmful (such as writing to the local disk), check for permission. In Java releases prior to 1.2, the methods of the Java API checked permission by querying the *security manager*. The security manager is a special object that defines a custom security policy for the application.

In Java 1.2, the job of the security manager was taken over by the *access controller*, a class that performs stack inspection to determine whether the operation should be allowed. (For backwards compatibility, the security manager still exists in Java 1.2.) By enforcing the security policy established by the security manager and access controller, the Java API helps to establish a safe environment in which potentially unsafe code can run.

API reference to the "fundamental classes" in the Java programming environment. The fundamental classes in the Java Development Kit (JDK) provide a powerful set of tools for creating portable applications; they are an important component of the toolbox used by every Java programmer. This reference covers the classes in the `java.lang`, `java.io`, `java.net`, `java.util`, `java.lang.reflect` packages.

But in the space context, not all of them are required; note that the material herein does not cover the classes that comprise the AWT and Swing graphics, such as the classes in the `java.math` (BigInteger class, not the same APIs that java.lang.math !) `java.text`, `java.util.zip java.rmi`, `java.sql`, and `java.security` packages.

3.2.1 General requirements

Basic requirements

REQ/AERO.API.GEN.0010	The full java core language without restriction shall be available to write embedded application code. *Remarks :*Core language mean the part of the language independent of APIs.		STD	T
REQ/AERO.API.GEN.0020	The following standard Java API shall be support with restriction detailed in next chapter : o java/io o java/lang o java/lang/ref o java/lang/reflect o java/net o java/util o javax/realtime		Aero	T
REQ/AERO.API.GEN.0030	With supported APIs is defined the minimum supported APIs by the AERO JVM. Supported mean that application could use this APIs as required, but not involve to necessary embed all APIs if they're not all required. Only a set of required APIs (take in the minimum supported APIs) could be embed.		Aero	

REQ/AERO.API.GEN.0040	The following set of data types shall be provided : - Boolean, - Integer (on 32 bits), - Double integer precision (on 64 bits), - Floating point, - Double floating point precision (on 64 bits), - Multi-dimension arrays		STD	T
REQ/AERO.API.GEN.0045	An java application shall be able to compute complex mathematical operations on integer, floating point, long and double values.		STD	T

3.2.2 Standard API supported in embedded context

3.2.2.1 I/O API

The package `java.io` contains the classes that handle fundamental input and output operations in Java. The I/O classes can be grouped as follows:

- Classes for reading input from a stream of data.
- Classes for writing output to a stream of data.
- Classes that manipulate files on the local filesystem.
- Classes that handle object serialization.

I/O in Java is based on streams. A stream represents a flow of data or a channel of communication. Java 1.0 supports only byte streams. The `InputStream` class is the superclass of all of the Java 1.0 byte input streams, while `OutputStream` is the superclass of all the byte output streams. The drawback to these byte streams is that they do not always handle Unicode characters correctly.

As of Java 1.1, `java.io` contains classes that represent character streams. These character stream classes handle Unicode characters appropriately by using a character encoding to convert bytes to characters and vice versa. The `Reader` class is the superclass of all the Java 1.1 character input streams, while `Writer` is the superclass of all character output streams.

The `InputStreamReader` and `OutputStreamWriter` classes provide a bridge between byte streams and character streams. By wrapping an `InputStreamReader` around an `InputStream` object, the bytes in the byte stream are read and converted to characters using the character encoding scheme specified by the `InputStreamReader`. Likewise, it is possible to wrap an `OutputStreamWriter` around any `OutputStream` object so that it is possible to write characters and have them converted to bytes.

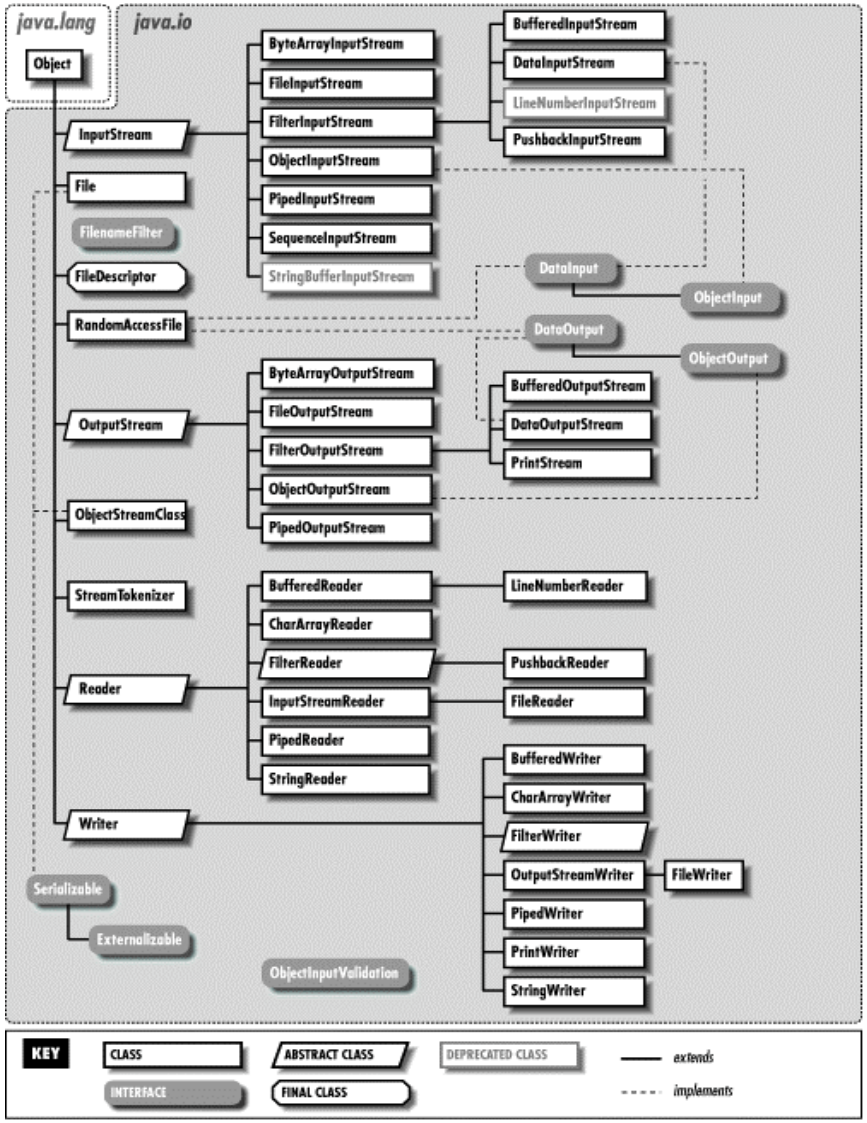
As of Java 1.1, `java.io` also contains classes to support object serialization. Object serialization is the ability to write the complete state of an object to an output stream, and then later recreate that object by reading in the serialized state from an

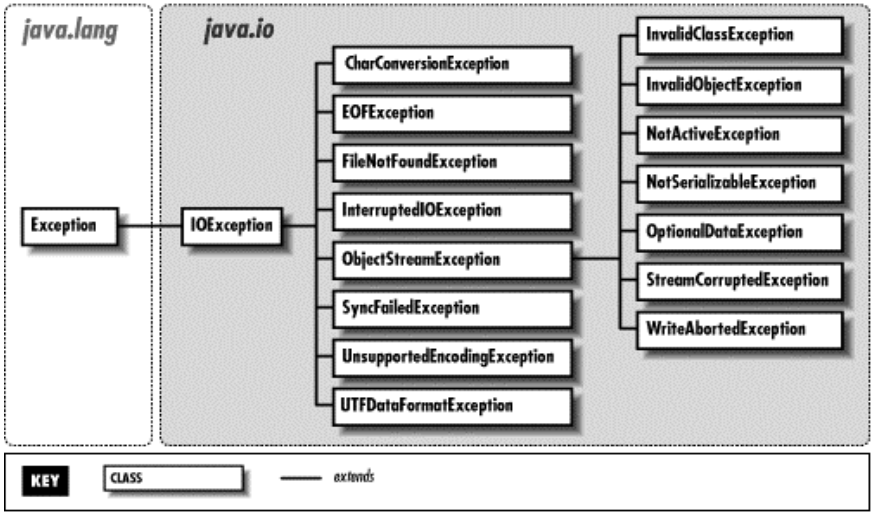
input stream. The `ObjectOutputStream` and `ObjectInputStream` classes handle serializing and deserializing objects, respectively.

The `RandomAccessFile` class is the only class that does not use a stream for reading or writing data. As its name implies, `RandomAccessFile` provides nonsequential access to a file for both reading and writing purposes.

The `File` class represents a file on the local file system. The class provides methods to identify and retrieve information about a file.

Next figures shows the class hierarchy for the `java.io` package. The `java.io` package defines a number of standard I/O exception classes. These exception classes are all subclasses of `IOException`, as shown in next figures.


java.lang
java.io
Object
InputStream
File
FilenameFilter
FileDescriptor
RandomAccessFile
OutputStream
ObjectStreamClass
StreamTokenizer
Reader
Writer
Serializable
Externalizable
ObjectInputValidation
ByteArrayInputStream
FileInputStream
FilterInputStream
ObjectInputStream
PipedInputStream
SequenceInputStream
StringBufferInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
PushbackInputStream
DataInput
ObjectInput
DataOutput
ObjectOutput
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
BufferedOutputStream
DataOutputStream
PrintStream
BufferedReader
CharArrayReader
FilterReader
InputStreamReader
PipedReader
StringReader
LineNumberReader
PushbackReader
FileReader
BufferedWriter
CharArrayWriter
FilterWriter
OutputStreamWriter
PipedWriter
PrintWriter
StringWriter
FileWriter
KEY
CLASS
ABSTRACT CLASS
DEPRECATED CLASS
extends
INTERFACE
FINAL CLASS
implements


java.lang
java.io
Exception
IOException
CharConversionException
EOFException
FileNotFoundException
InterruptedIOException
ObjectStreamException
SyncFailedException
UnsupportedEncodingException
UTFDataFormatException
InvalidClassException
InvalidObjectException
NotActiveException
NotSerializableException
OptionalDataException
StreamCorruptedException
WriteAbortedException
KEY
CLASS
extends

Details of standard Java I/O API that shall be supported without restriction.

The following list of API defines the minimum requirement for supported APIs by the core JVM. Embedded APIs shall be taken into this list without require to take all (except dependable sub APIs).

REQ/AERO.API.GEN.0050	The java.io subset of APIs that the AERO JVM shall support without restriction is defined as :
	
	`java/io/BufferedInputStream`
	`java/io/BufferedOutputStream`
	`java/io/BufferedWriter`
	`java/io/ByteArrayInputStream`
	`java/io/ByteArrayOutputStream`
	`java/io/CharArrayReader`
	`java/io/CharArrayWriter`
	`java/io/CharConversionException`
	`java/io/DataInput`
	`java/io/DataInputStream`
	`java/io/DataOutput`
	`java/io/DataOutputStream`
	`java/io/EOFException`
	`java/io/Externalizable`
	`java/io/FileDescriptor`
	`java/io/FileFilter`
	`java/io/FileNotFoundException`
	`java/io/FileReader`
	`java/io/FileWriter`
	`java/io/FilenameFilter`
	`java/io/FilterInputStream`
	`java/io/FilterOutputStream`
	`java/io/FilterReader`
	`java/io/FilterWriter`
	`java/io/IOException`
	`java/io/InputStream`
	`java/io/InterruptedIOException`
	`java/io/InvalidClassException`
	`java/io/InvalidObjectException`
	`java/io/LineNumberInputStream`
			Aero	T

	java/io/LineNumberReader java/io/NotActiveException java/io/NotSerializableException java/io/ObjectInput java/io/ObjectInputValidation java/io/ObjectOutput java/io/ObjectStreamException java/io/OptionalDataException java/io/OutputStream java/io/PipedReader java/io/PipedWriter java/io/PushbackInputStream java/io/PushbackReader java/io/SequenceInputStream java/io/Serializable java/io/SerializablePermission java/io/StreamCorruptedException java/io/SyncFailedException			

Details of standard Java API that shall be supported with restriction (not all function defined in each APIs are required)

REQ/AERO.API.GEN.0060	The java.io subset of APIs that the AERO JVM shall support with restriction is defined as :
	
	`java/io/BufferedReader(1)`
	`java/io/File(1)`
	`java/io/FileInputStream(1)`
	`java/io/FileNotFoundException`
	`java/io/FileOutputStream(1)`
	`java/io/FilePermission(1)`
	`java/io/InputStreamReader(1)`
	`java/io/ObjectInputStream(1)`
	`java/io/ObjectOutputStream(1)`
	`java/io/ObjectStreamClass(1)`
	`java/io/ObjectStreamConstants(1)`
	`java/io/ObjectStreamField(1)`
	`java/io/OutputStreamWriter(1)`
	`java/io/PipedInputStream(1)`
	`java/io/PipedOutputStream(1)`
	`java/io/Reader(1)`
	`java/io/StreamTokenizer(1)`
			Aero	T

(1) exact subset of APIs will be defined in Detailed Design Document.

3.2.2.2 Lang API

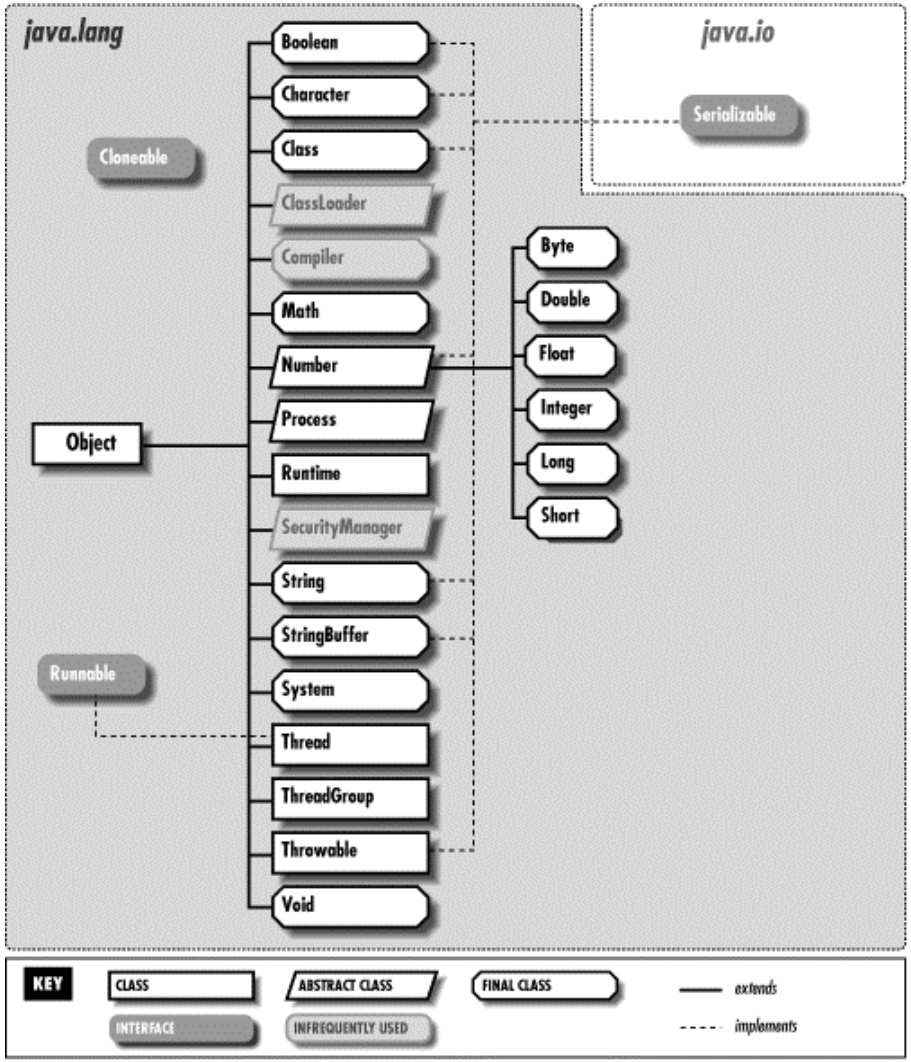
The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

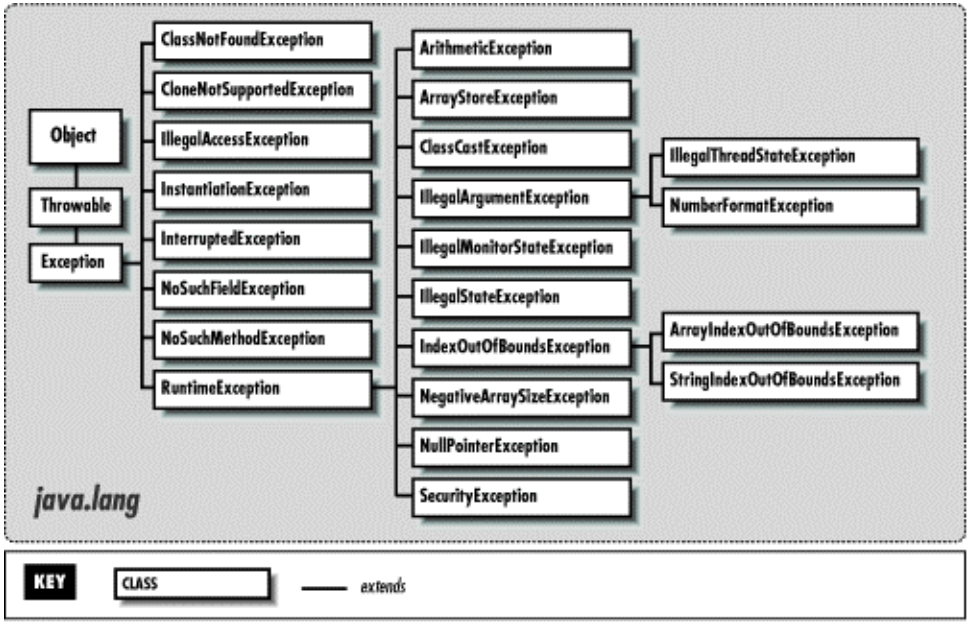
- `Object`, the ultimate superclass of all classes in Java.
- `Thread`, the class that controls each thread in a multithreaded program.
- `Throwable`, the superclass of all error and exception classes in Java.
- Classes that encapsulate the primitive data types in Java.
- Classes for accessing system resources and other low-level entities.
- `Math`, a class that provides standard mathematical methods.
- `String`, the class that represents strings.

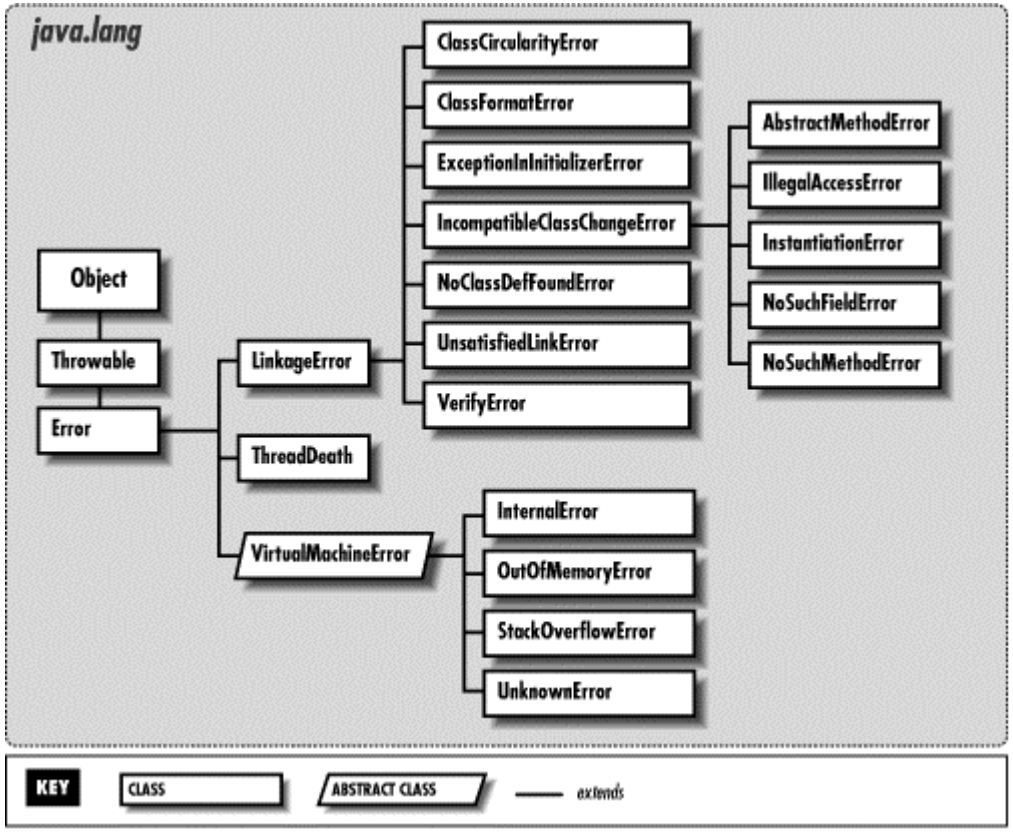
Because the classes in the `java.lang` package are so essential, the `java.lang` package is implicitly imported by every Java source file. In other words, it could be possible to all of the classes and interfaces in `java.lang` using their simple names.

Next figures shows the class hierarchy for the `java.lang` package.

The possible exceptions in a Java program are organized in a hierarchy of exception classes. The `Throwable` class is at the root of the exception hierarchy. `Throwable` has two immediate subclasses: `Exception` and `Error`. Next figures shows the standard exception classes defined in the `java.lang` package, and the standard error classes defined in `java.lang`.


java.lang
java.io
Boolean
Character
Class
ClassLoader
Compiler
Math
Number
Process
Runtime
SecurityManager
String
StringBuffer
System
Thread
ThreadGroup
Throwable
Void
Byte
Double
Float
Integer
Long
Short
Cloneable
Serializable
Object
Runnable
KEY
CLASS
ABSTRACT CLASS
FINAL CLASS
INTERFACE
INFREQUENTLY USED
extends
implements


Object
Throwable
Exception
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchFieldException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
IllegalThreadStateException
NumberFormatException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
java.lang
KEY
CLASS
extends


java.lang
Object
Throwable
Error
LinkageError
ThreadDeath
VirtualMachineError
ClassCircularityError
ClassFormatError
ExceptionInInitializerError
IncompatibleClassChangeError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
KEY
CLASS
ABSTRACT CLASS
extends

Details of standard Java Lang API that shall be supported without restriction.

The following list of API defines the minimum requirement for supported APIs by the core JVM. Embedded APIs shall be taken into this list without require to take all (except dependable sub APIs).

REQ/AERO.API.GEN.0070	The java.lang subset of APIs that the AERO JVM shall support without restriction is defined as :
	
	`java/lang/AbstractMethodError`
	`java/lang/ArithmeticException`
	`java/lang/ArrayIndexOutOfBoundsException`
	`java/lang/ArrayStoreException`
	`java/lang/Boolean`
	`java/lang/Byte`
	`java/lang/ClassCastException`
	`java/lang/ClassCircularityError`
	`java/lang/ClassFormatError`
	`java/lang/ClassNotFoundException`
	`java/lang/CloneNotSupportedException`
	`java/lang/Cloneable`
	`java/lang/Comparable`
	`java/lang/Compiler`
	`java/lang/Double`
	`java/lang/Error`
	`java/lang/Exception`
	`java/lang/ExceptionInInitializerError`
	`java/lang/Float`
	`java/lang/IllegalAccessError`
	`java/lang/IllegalAccessException`
	`java/lang/IllegalArgumentException`
	`java/lang/IllegalMonitorStateException`
	`java/lang/IllegalStateException`
	`java/lang/IllegalThreadStateException`
	`java/lang/IncompatibleClassChangeError`
	`java/lang/IndexOutOfBoundsException`
	`java/lang/InstantiationError`
	`java/lang/InstantiationException`
	`java/lang/Integer`
			Aero	T

	java/lang/InternalError java/lang/InterruptedException java/lang/LinkageError java/lang/Long java/lang/Math java/lang/NegativeArraySizeException java/lang/NoClassDefFoundError java/lang/NoSuchFieldError java/lang/NoSuchFieldException java/lang/NoSuchMethodError java/lang/NoSuchMethodException java/lang/NullPointerException java/lang/Number java/lang/NumberFormatException java/lang/Object java/lang/OutOfMemoryError java/lang/Process java/lang/Runnable java/lang/RuntimeException java/lang/RuntimePermission java/lang/SecurityException java/lang/Short java/lang/StackOverflowError java/lang/ThreadDeath java/lang/ThreadLocal java/lang/UnknownError java/lang/UnsatisfiedLinkError java/lang/UnsupportedClassVersionError java/lang/UnsupportedOperationException java/lang/VerifyError java/lang/VirtualMachineError java/lang/Void java/lang/ref/PhantomReference java/lang/ref/Reference java/lang/ref/ReferenceQueue java/lang/ref/SoftReference java/lang/ref/WeakReference			

Details of standard Java API that shall be supported with restriction (not all function defined in each APIs are required)

REQ/AERO.API.GEN.0080	The java.io subset of APIs that the AERO JVM shall support with restriction is defined as :
	
	`java/lang/Class(1)`
	`java/lang/ClassLoader(1)`
	`java/lang/InheritableThreadLocal(1)`
	`java/lang/Package(1)`
	`java/lang/Runtime(1)`
	`java/lang/SecurityManager(1)`
	`java/lang/System(1)`
	`java/lang/Thread(1)`
	`java/lang/ThreadGroup(1)`
	`java/lang/Throwable(1)`
	`java/lang/reflect/AccessibleObject(1)`
	`java/lang/reflect/Constructor(1)`
	`java/lang/reflect/Method(1)`
			Aero	T

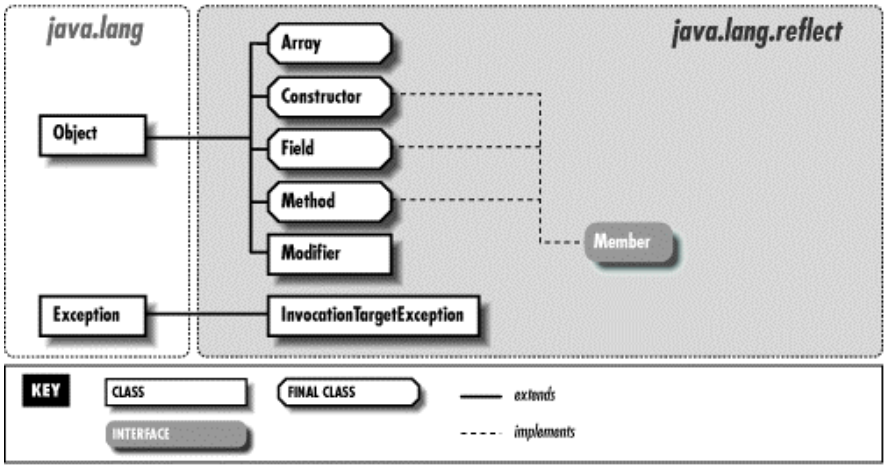
(1) exact subset of APIs will be defined in Detailed Design Document.

3.2.2.3 Lang/Reflect API

The package `java.lang.reflect` is new as of Java 1.1. It contains classes and interfaces that support the Reflection API. Reflection refers to the ability of a class to reflect upon itself, or look inside of itself, to see what it can do. The Reflection API makes it possible to:

- Discover the variables, methods, and constructors of any class.
- Create an instance of any class using any available constructor of that class, even if the class initiating the creation was not compiled with any information about the class to be instantiated.
- Access the variables of any object, even if the accessing class was not compiled with any information about the class to be accessed.
- Call the methods of any object, even if the calling class was not compiled with any information about the class that contains the methods.
- Create an array of objects that are instances of any class, even if the creating class was not compiled with any information about the class.

These capabilities are implemented by the `java.lang.Class` class and the classes in the `java.lang.reflect` package. Next figure shows the class hierarchy for the `java.lang.reflect` package.



Java 1.1 currently uses the Reflection API for two purposes:

- The JavaBeans API supports a mechanism for customizing objects that is based on being able to discover their public variables, methods, and constructors. JavaBeans are not foreseen to be embedded in space context.
- The object serialization functionality in `java.io` is built on top of the Reflection API. Object serialization allows arbitrary objects to be written to a stream of bytes and then read back later as objects.

Space context could use the Reflection to develop new onboard capabilities to investigate when error occurs in embedded application code, monitoring data etc.

Details of standard Java Reflect API that shall be supported without restriction.

The following list of API defines the minimum requirement for supported APIs by the core JVM. Embedded APIs shall be taken into this list without require to take all (except dependable sub APIs).

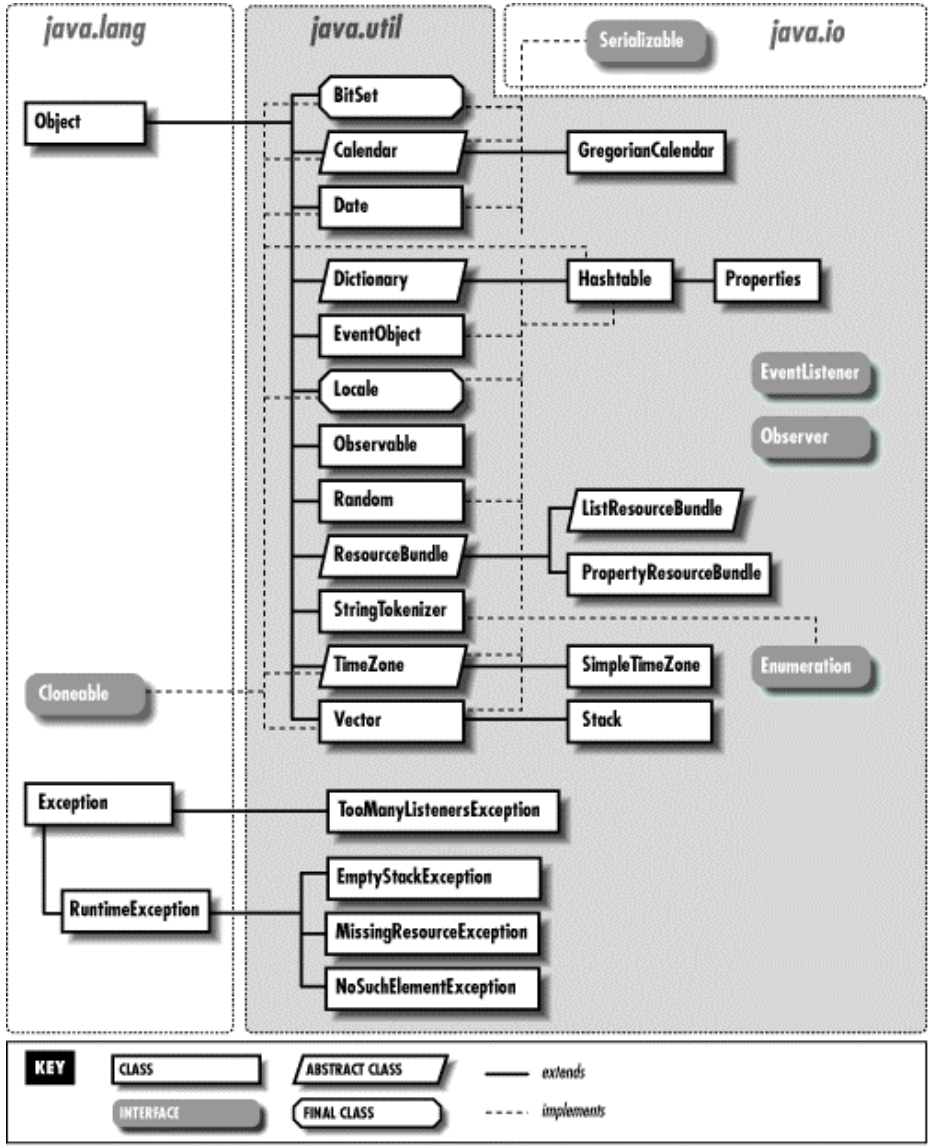
REQ/AERO.API.GEN.0090	The java.lang.reflect subset of APIs that the AERO JVM shall support without restriction is defined as :
	
	`java/lang/reflect/Array`
	`java/lang/reflect/Field`
	`java/lang/reflect/InvocationTargetException`
	`java/lang/reflect/Member`
	`java/lang/reflect/Modifier`
	`java/lang/reflect/ReflectPermission`
			Aero	T

3.2.2.4 Util API

The package `java.util` contains a number of useful classes and interfaces. Although the name of the package might imply that these are utility classes, they are really more important than that. In fact, Java depends directly on several of the classes in this package, and many programs will find these classes indispensable. The classes and interfaces in `java.util` include:

- The `Hashtable` class for implementing hashtables, or associative arrays.
- The `Vector` class, which supports variable-length arrays.
- The `Enumeration` interface for iterating through a collection of elements.
- The `StringTokenizer` class for parsing strings into distinct tokens separated by delimiter characters.
- The `EventObject` class and the `EventListener` interface, which form the basis of the new AWT event model in Java 1.1.
- The `Locale` class in Java 1.1, which represents a particular locale for internationalization purposes.
- The `Calendar` and `TimeZone` classes in Java. These classes interpret the value of a `Date` object in the context of a particular calendar system.
- The `ResourceBundle` class and its subclasses, `ListResourceBundle` and `PropertyResourceBundle`, which represent sets of localized data in Java 1.1.

Next figure shows the class hierarchy for the `java.util` package.


java.lang
java.util
java.io
Serializable
Object
BitSet
Calendar
GregorianCalendar
Date
Dictionary
Hashtable
Properties
EventObject
EventListener
Locale
Observer
Observable
Random
ListResourceBundle
ResourceBundle
PropertyResourceBundle
StringTokenizer
Enumeration
TimeZone
SimpleTimeZone
Cloneable
Vector
Stack
Exception
TooManyListenersException
EmptyStackException
RuntimeException
MissingResourceException
NoSuchElementException
KEY
CLASS
ABSTRACT CLASS
extends
INTERFACE
FINAL CLASS
implements

Details of standard Java Util API that shall be supported without restriction.

The following list of API defines the minimum requirement for supported APIs by the core JVM. Embedded APIs shall be taken into this list without require to take all (except dependable sub APIs).

REQ/AERO.API.GEN.0100	The java.util subset of APIs that the AERO JVM shall support without restriction is defined as :
	
	`java/util/AbstractCollection`
	`java/util/AbstractMap`
	`java/util/AbstractSequentialList`
	`java/util/AbstractSet`
	`java/util/ArrayList`
	`java/util/Arrays`
	`java/util/BitSet`
	`java/util/Collection`
	`java/util/Collections`
	`java/util/Comparator`
	`java/util/ConcurrentModificationException`
	`java/util/Dictionary`
	`java/util/EmptyStackException`
	`java/util/Enumeration`
	`java/util/EventListener`
	`java/util/EventObject`
	`java/util/HashMap`
	`java/util/HashSet`
	`java/util/Hashtable`
	`java/util/Iterator`
	`java/util/LinkedList`
	`java/util/List`
	`java/util/ListIterator`
	`java/util/Map`
	`java/util/MissingResourceException`
	`java/util/NoSuchElementException`
	`java/util/Observable`
	`java/util/Observer`
	`java/util/Properties`
	`java/util/Random`
			Aero	T

	java/util/ResourceBundle java/util/Set java/util/SortedMap java/util/SortedSet java/util/Stack java/util/StringTokenizer java/util/TooManyListenersException java/util/Vector java/util/WeakHashMap			

Details of standard Java API that shall be supported with restriction (not all function defined in each APIs are required)

REQ/AERO.API.GEN.0110	The java.util subset of APIs that the AERO JVM shall support with restriction is defined as : java/util/AbstractList(1) java/util/Calendar(1) java/util/Date(1) java/util/jar/Attributes(1) java/util/PropertyPermission(1) java/util/PropertyResourceBundle(1) java/util/GregorianCalendar(1) java/util/ListResourceBundle(1) java/util/Locale(1) java/util/jar/JarEntry(1) java/util/jar/JarFile(1) java/util/jar/JarInputStream(1) java/util/jar/Manifest(1)		Aero	T

(1) exact subset of APIs will be defined in Detailed Design Document.

3.2.3 Specific new API in embedded context

3.2.3.1 javax.realtime API

The package `javax.realtime` is a new package introduced by the RTJ group and specified through the RTSJ document. It contains a number of useful classes and interfaces specific for real-time.

The classes and interfaces in `javax.realtime` include:

- The `AsyncEvent` class for implementing Asynchronous event and timer
- The `MemoryArea` class, which supports different memory types.
- The `Monitor` class for priority management.
- The `MemoryControl` class for memory management
- The `RealtimeSecurity` class and the `RealtimeSystem` class, which form the basis of the new realtime model in Java.
- The `Scheduler` class which represents a upgrade of the base Java scheduler.
- The `RealTimeThread` extension of base Thread.
- The `Throwable` extension class and its subclasses which represent sets of new Error and Exception for realtime

The complete javax.realtime hierarchy class is provided in annex.

The full standard Java realtime API defined in RTJS shall be supported without restriction. Embedded APIs shall be taken into this list without require to take all (except dependable sub APIs).

REQ/AERO.API.GEN.0120	The AERO JVM shall support API specified in the final release V1.0 of 11/12/2001 of javax.realtime API. It may include later additions and clarification.		RTSJ	T

3.2.3.2 Others APIs

Some new generic functions will be necessary to implement onboard application, even if representative application porting in Java will show later API that could be defined, some first basic functions could be defined.

REQ/AERO.API.GEN.0130	An application shall be able to send internal message to the rest of onboard software through a unique simple mechanism.		Aero	T
REQ/AERO.API.GEN.0140	An application shall be able to send internal message to the rest of onboard software and wait an acknowledge through a unique simple mechanism		Aero	T
REQ/AERO.API.GEN.0150	An application shall be able to read on board time with a precision of [TBD – implantation depending of On-board software]		Aero	T
REQ/AERO.API.GEN.0160	An application shall be able to wait a specified on board time with a precision of [TBD – implantation depending of On-board software].		Aero	T

3.2.4 API for test & debug purposes

This chapter details of standard Java API subset that shall be supported for debug and test purposes (not in embedded context). The following list of API defines the minimum requirement for supported APIs by the core JVM. This APIs will be used for validation and debug of application, but are not destined to be embed.

3.2.4.1 I/O API

REQ/AERO.API.GEN.0170	The java.io subset of APIs that the AERO JVM shall support for test purposes without restriction is defined as :
	
	`java/io/PrintStream`
	`java/io/PrintWriter`
	`java/io/RandomAccessFile`
	`java/io/StringBufferInputStream`
	`java/io/StringReader`
	`java/io/StringWriter`
	`java/io/UTFDataFormatException`
	`java/io/UnsupportedEncodingException`
	`java/io/WriteAbortedException`
	`java/io/Writer`
			Aero	T

3.2.4.2 Lang API

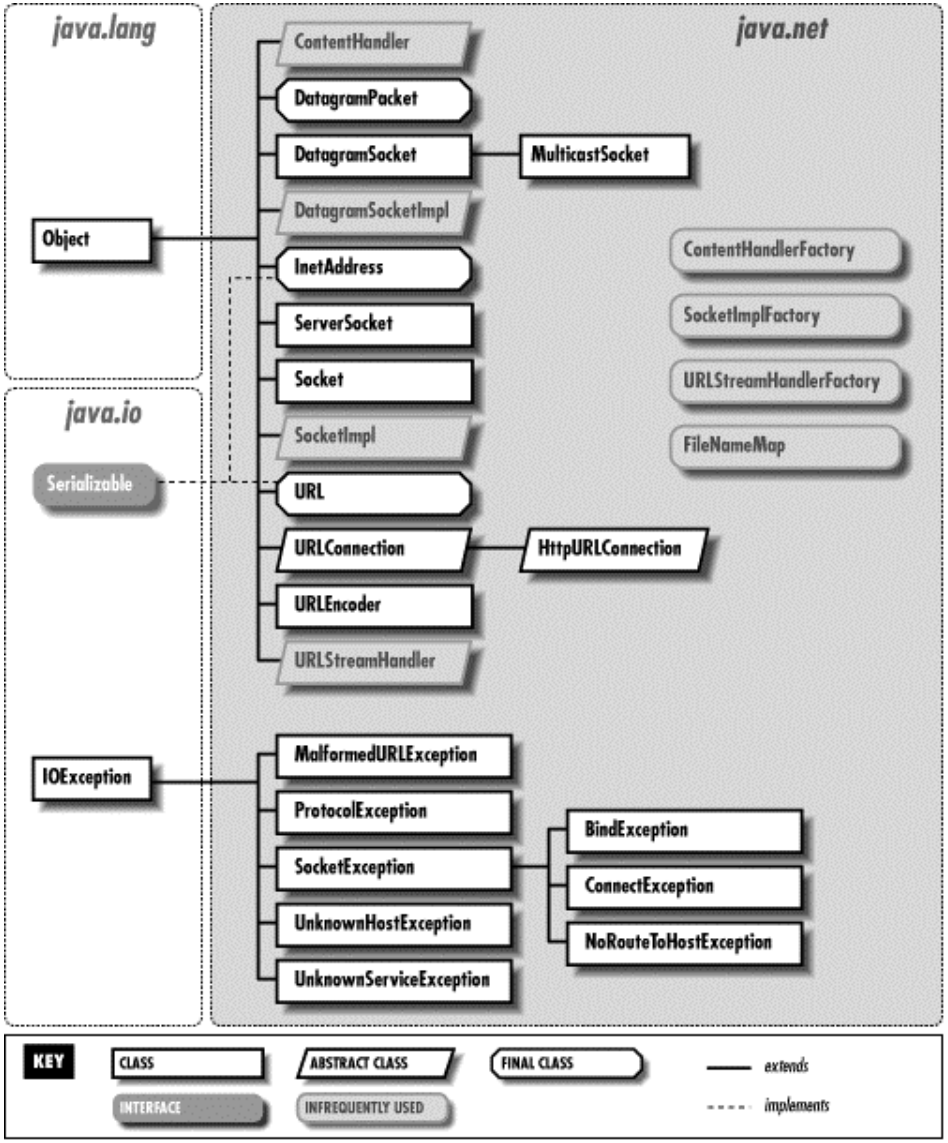
REQ/AERO.API.GEN.0180	The java.lang subset of APIs that the AERO JVM shall support for test purposes without restriction is defined as :
	
	`java/lang/Character`
	`java/lang/String`
	`java/lang/StringBuffer`
	`java/lang/StringIndexOutOfBoundsException`
			Aero	T

3.2.4.3 Net API

The package `java.net` contains classes and interfaces that provide a powerful infrastructure for networking in Java. These include:

- The `URL` class for basic access to Uniform Resource Locators (URLs).
- The `URLConnection` class, which supports more complex operations on URLs.
- The `Socket` class for connecting to particular ports on specific Internet hosts and reading and writing data using streams.
- The `ServerSocket` class for implementing servers that accept connections from clients.
- The `DatagramSocket`, `MulticastSocket`, and `DatagramPacket` classes for implementing low-level networking.
- The `InetAddress` class, which represents Internet addresses.

Next figure shows the class hierarchy for the `java.net` package.


java.lang
java.net
java.io
Object
Serializable
IOException
ContentHandler
DatagramPacket
DatagramSocket
MulticastSocket
DatagramSocketImpl
InetAddress
ServerSocket
Socket
SocketImpl
URL
URLConnection
HttpURLConnection
URLEncoder
URLStreamHandler
ContentHandlerFactory
SocketImplFactory
URLStreamHandlerFactory
FileNameMap
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException
BindException
ConnectException
NoRouteToHostException
KEY
CLASS
ABSTRACT CLASS
FINAL CLASS
INTERFACE
INFREQUENTLY USED
extends
implements

REQ/AERO.API.GEN.0190	The java.net subset of APIs that the AERO JVM shall support for test purposes without restriction is defined as :
	
	`java/net/Authenticator`
	`java/net/BindException`
	`java/net/ConnectException`
	`java/net/ContentHandler`
	`java/net/ContentHandlerFactory`
	`java/net/DatagramPacket`
	`java/net/DatagramSocket`
	`java/net/DatagramSocketImpl`
	`java/net/FileNameMap`
	`java/net/HttpURLConnection`
	`java/net/InetAddress`
	`java/net/JarURLConnection`
	`java/net/MalformedURLException`
	`java/net/MulticastSocket`
	`java/net/NetPermission`
	`java/net/NoRouteToHostException`
	`java/net/PasswordAuthentication`
	`java/net/ProtocolException`
	`java/net/ServerSocket`
	`java/net/Socket`
	`java/net/SocketException`
	`java/net/SocketImpl`
	`java/net/SocketImplFactory`
	`java/net/SocketInputStream`
	`java/net/SocketOptions`
	`java/net/SocketOutputStream`
	`java/net/SocketPermission`
			Aero	T

3.3 JNI

The Java Native Interface (JNI) is a standard mechanism for inter-operability between Java and native code, i.e., code written in non-portable system programming languages like C. Last standard release is the version 1.2 of the Java Native Interface. Since JNI provides hundreds of routines, including support for JNI would pose too big an overhead on those applications that are not using it. This is why JNI support must be activated explicitly when required.

Native code that is interfaced through the JNI interface is typically stored in shared libraries that are dynamically loaded by the virtual machine when the application uses native code. Since dynamically loading libraries is not possible on small embedded systems that don't provide a file system, a different approach must be taken. Instead of loading a library, it is preferable to have the native code be part of the application itself, i.e., to link the native object code directly with the application

REQ/AERO.API.GEN.0200	AERO JVM shall provides the support for JNI in a Real-Time deterministic implementation		Aero	T
REQ/AERO.API.GEN.0210	It shall be possible to specify an option that enables the support for JNI		Aero	I
REQ/AERO.API.GEN.0220	It shall be possible to allows direct linking of native object code with the application through an option. This option could be used in addition to the JNI support option		Aero	I
REQ/AERO.API.GEN.0230	To build an application that uses the native code on a target that requires manual linking, it might also be required to provide these object files to the linker, and it might even be required to provide a specific object file that contains the JNI support		Aero	I

4. Environment requirements

4.1 Tools

REQ/AERO.ENVIRON.010	A tool for creating a single executable image out of the AERO JVM and a set of Java classes. shall be develop. This image can be loaded into flash-memory or ROM, avoiding the need for a file-system in the target platform. This tool shall be extended such that opportunities to replace dynamic allocation by static allocation whenever analysis of the application reveals that this optimisation is possible.		Aero	T
REQ/AERO.ENVIRON.020	The static GC shall be part of the tool to build an executable image. It shall bring fast and predictable execution to the affected heap operations.		Aero	I
REQ/AERO.ENVIRON.030	A simulator of the AERO JVM shall be provided by porting the VM to the future application development system's OS (Solaris or Windows) to be able to run applications there. Native code (e.g. accessing hardware) shall not be included in the emulation Also, Java code that accesses hardware directly (eg. through RTSJ's PhysicalMemory class) will not work directly on such an emulation and stub will be written [TBC]		Aero	T
REQ/AERO.ENVIRON.040	For most effective memory usage, a tool that finds the amount of memory that is actually used by an application shall be provided. This allows for exact selection of the memory required for the system and to select a heap size for optimal run-time performance.		Aero	T

4.2 Operating System

REQ/AERO.ENVIRON.050	AERO JVM software shall run on Sparc ERC32 processor, under a standard posix Operating System		Aero	I
REQ/AERO.ENVIRON.060	AERO JVM environment tools shall run on Solaris Operating System and/or Linux.		Aero	I

5. Operability requirements

5.1 User's manual

REQ/AERO.OPERAB.010	A user ’s manual shall be written		Aero	I

5.2 On line help

N/A

5.3 Interface standard

N/A

5.4 Interface ergonomy

N/A

6. Development requirements

REQ/AERO.DEVELOP.010	The tailored ECSS-B standard provided in [MP] is applicable to development.		Aero	I/T

7. Portability and maintainability requirements

7.1 Portability of design and code

REQ/AERO.PORT.010	The solution shall be implemented in ANSI C using the GNU gcc cross compiler		Aero	I
REQ/AERO.PORT.020	Threads shall be based on the POSIX threads standard.		STD	I
REQ/AERO.PORT.030	Design shall provide clear separation of platform-dependant from platform-independant code to reduces the required effort to port to other platforms.		Aero	I
REQ/AERO.PORT.040	It shall be possible to port the solution to a new posix operating system..		Aero	I

7.2 Maintainability requirements

REQ/AERO.MAINT.010	The source code shall be delivered to Astrium to be analysed by Astrium quality engineers : metrics will be made on code, especially size of modules, cyclomatic complexity.		Aero	N/A
REQ/AERO.MAINT.020	The implementation is required to provide in DDD a documentation stating exactly the algorithm used for granting such access of thread when they're are preempted in favor of a thread with higher execution eligibility.		Aero	I
REQ/AERO.MAINT.030	The implementation is required to provide in the DDD a documentation stating exactly the algorithm used for such placement of threads, with higher priority than preempted ones, that may be given access to the processor at any time as determined by a particular implementation.		Aero	I
REQ/AERO.MAINT.040	Implementations that provide a monitor control algorithm in addition to those described in this		Aero	I

	document are required to clearly document the behavior of that algorithm in DDD.			

8.

CHAPTER REMOVED
FROM THE AERO PROJECT
ORIGINAL SP1 DOCUMENT