The Real-time Specification for Java

Roadmap

- **Overview of the RTSJ**
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control
- Schedulability Analysis
- Conclusions

The Real-Time Specification for Java

Lecture aims

- To give the background of the RTSJ and the NIST requirements
- To provide an introduction to
  - Memory management
  - Time values and clocks
  - Schedulable objects and scheduling

Background and NIST Requirements

- In the late 1990s, the US National Institute of Standards and Technology (NIST) coordinated the derivation of guiding principles and requirements for real-time extensions to Java
- Requirements
  - Fixed priority and round robin scheduling
  - Mutual exclusion locking (avoiding priority inversion)
  - Inter-thread communication (e.g. semaphores)
  - User-defined interrupt handlers and device drivers — including the ability to manage interrupts (e.g., enabling and disabling)
  - Timeouts and aborts on running threads
  - Profiles to cope with the variety of applications, these included: safety critical, no dynamic loading, and distributed real-time

Implementations must provide

- A framework for finding available profiles
- Bounded pre-emption latency on any garbage collection
- A well-defined model for real-time Java threads
- Communication and synchronization between real-time and non real-time threads
- Mechanisms for handling internal and external asynchronous events
- Asynchronous thread termination
- Mutual exclusion without blocking
- The ability to determine whether the running thread is real-time or non real-time
- A well-defined relationship between real-time and non real-time threads

RTSJ Guiding Principles

- Be backward compatible with non real-time Java programs
- Support the principle of "Write Once, Run Anywhere" but not at the expense of predictability
- Address the current real-time system practice and allow future implementations to include advanced features
- Give priority to predictable execution in all design trade-offs
- Require no syntactic extensions to the Java language
- Allow implementers flexibility

Overview of Enhancements

- The RTSJ enhances Java in the following areas:
  - memory management
  - time values and clocks
  - schedulable objects and scheduling
  - real-time threads
  - asynchronous event handling and timers
  - asynchronous transfer of control
  - synchronization and resource sharing
  - physical memory access

Warning

The RTSJ only addresses the execution of real-time Java programs on single processor systems. It attempts not to preclude execution on a shared-memory multiprocessor systems but it has no support for, say, allocation of threads to processors.

Memory Management

- Many RTS have limited amount of memory available because of
  - the cost
  - constraints associated with surrounding system (size, power, weight)
- It is necessary to control how memory is allocated to use it effectively
- Where there is more than one type of memory (with different access characteristics), it may be necessary to instruct the compiler to place certain data types at certain locations
- By doing this, the program is able to increase performance and predictability as well as interact with the outside world

Heap Memory

- The JVM is responsible for managing the heap
- Problems are how much space is required and when to release it
- The latter can be handled in several ways, including
  - require the programmer to return the memory explicitly — this is error prone but is easy to implement
  - require the JVM to monitor the memory and release chunks which are no longer being used

Real-Time Garbage Collection

- From a RT perspective, the approaches have an impact on the ability to analyze the program's timing properties
- Garbage collection may be performed either when the heap is full or by an incremental activity
- In either case, running the garbage collector may have a significant impact on the response time of a time-critical thread
- Objects in standard Java are allocated on the heap and the language requires garbage collection
- The garbage collector runs as part of the JVM

Memory Areas

- RTSJ provides memory management which is not affected by GC
- It defines **memory areas**, some of which exist outside the traditional Java heap and never suffer garbage collection
- RTSJ requires that the GC be preemptible by real-time threads and that be a bounded latency for preemption
- The `MemoryArea` class is an abstract class from which for all RTSJ memory areas are derived
- When a particular memory area is entered, all object allocation is performed within that area

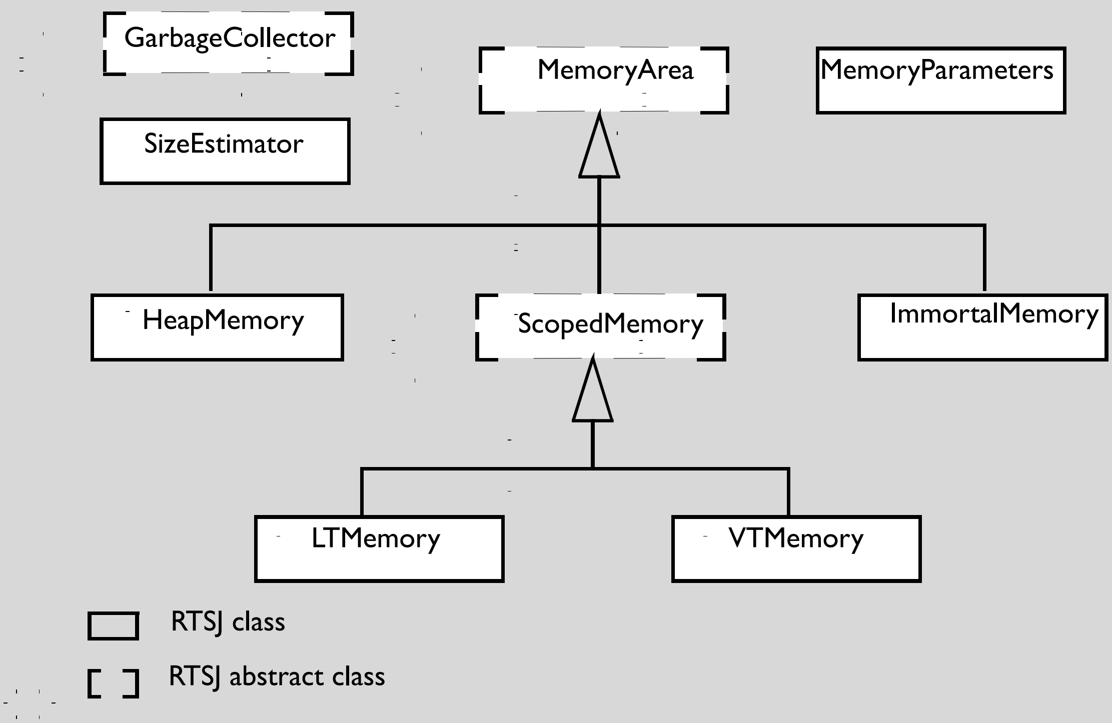
Subclasses of MemoryArea

- `HeapMemory` allows objects to be allocated in the heap
- `ImmortalMemory` is shared among all threads; objects created here are never subject to GC
- `ScopedMemory` is for objects that have a well-defined lifetime; with each scoped memory is a reference count which keeps track of how many real-time entities are currently using it
  - When the reference count goes from 1 to 0, all objects in the area have their finalization method executed and the memory is reclaimed
- `ScopedMemory` is abstract and has several subclasses
  - `VTMemory`: allocations may take variable amounts of time
  - `LTMemory`: allocations occur in linear time to the size of the object

Memory Parameters

- Can be given when real-time threads and asynchronous event handlers are created; they specify
  - maximum amount of memory a thread/handler can use in an area,
  - max. amount of memory that can be consumed in immortal memory
  - a limit on the rate of allocation from the heap (in bytes per second),
- Can be used by the scheduler as part of an admission control policy and/or for the purpose of ensuring adequate GC

Memory Management Classes



Time Values

- `HighResolutionTime` for time values with nanosecond granularity represented by a 64 bits milliseconds and a 32 bits nanoseconds component
- The class is abstract with subclasses:
  - `AbsoluteTime`: expressed as a time relative to some epoch. This epoch depends of the associated clock. It might be 1/1/970 for the wall clock or system start-up time for a monotonic clock
  - RelativeTime
- Time values are also relative to particular clocks

Clocks

- `Clock` is the abstract class from which all clocks are derived
- RTSJ allows many types of clocks; eg, there could be a CPU execution-time clock (although this is not required)
- At least one real-time clock which advances monotonically
- A static method **`getRealtimeClock`** returns this clock

Summary

- The RTSJ originates from the desire to use Java in real-time
- Java's main problems include unsuitable memory management because of GC and poor support for clocks and time
- The RTSJ stems from the NIST requirements
- Memory management is augmented by memory areas
- Clocks are augmented by a real-time clock and by high resolution absolute and relate time types

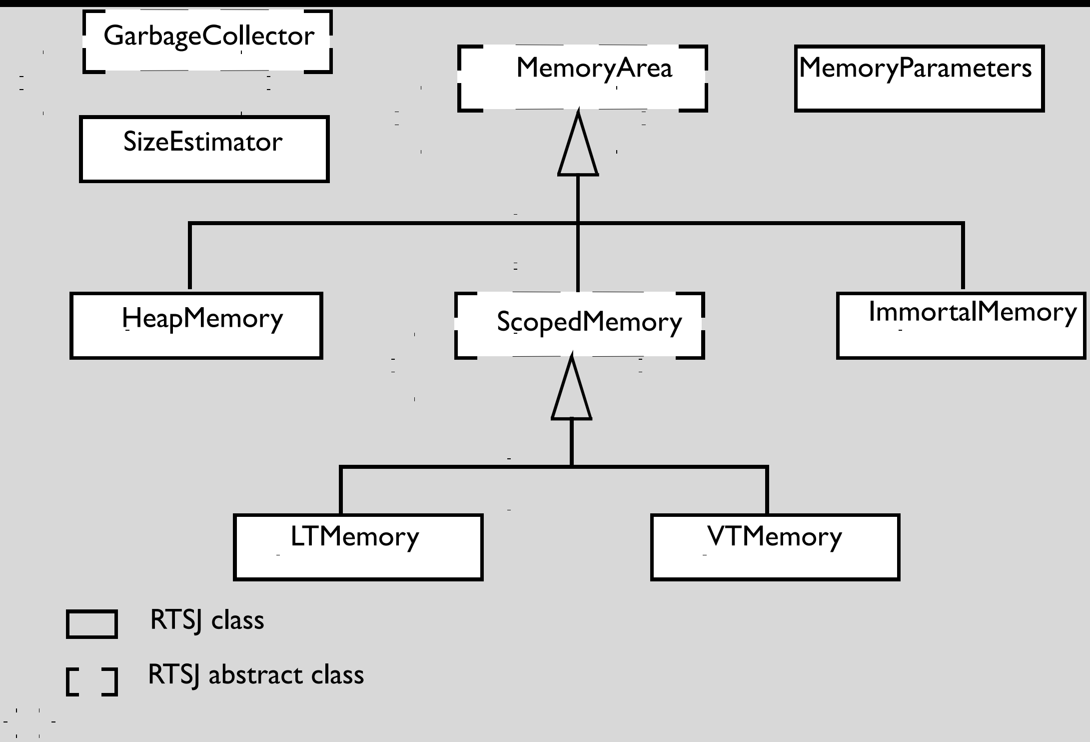
Roadmap

- **Overview of the RTSJ**
- Memory Management
- Clocks and Time
- Scheduling and Schedulable Objects
- Asynchronous Events and Handlers
- Real-Time Threads
- Asynchronous Transfer of Control
- Resource Control

The RTSJ Overview Continued

- Lecture aims
- To complete the overview of the RTSJ by considering
  - Schedulable objects and scheduling
  - Real-time threads
  - Asynchronous events and timers
  - Asynchronous transfer of control
  - Synchronization and resource sharing
  - Physical memory access

Memory Management Classes



Scheduling in Java

- Java offers no guarantees that the highest priority runnable thread will always be the one executing
- This is because the OS may not support preemptive priority-based scheduling
- Java only defines 10 priority levels and an implementation is free to map these priorities onto a more restricted host operating system's priority range if necessary
- The weak definition of scheduling and restricted range of priorities means Java lacks predictability and, hence, Java's use for RT systems implementation is severely limited

Schedulable Objects

- RTSJ generalizes the entities that can be scheduled from threads to the notion of schedulable objects
- A **schedulable object** implements the `Schedulable` interface
- Each schedulable object must also indicate its specific
  - release requirement (that is, when it should become runnable),
  - memory requirements (e.g., heap allocation rate)
  - scheduling requirements (e.g, priority at which it should be scheduled)

Release Parameters

- Scheduling theories often identify three types of releases:
  - periodic (released on a regular basis)
  - aperiodic (released at random)
  - sporadic (released irregularly but with a minimum time bw releases)
- All release parameters have:
  - cost: amount of cpu time needed every release
  - deadline: the time at which the current release must have finished
- `PeriodicParameters` also include the start time for the first release and the time interval (period) between releases.
- `SporadicParameter` include the minimum inter-arrival time between releases

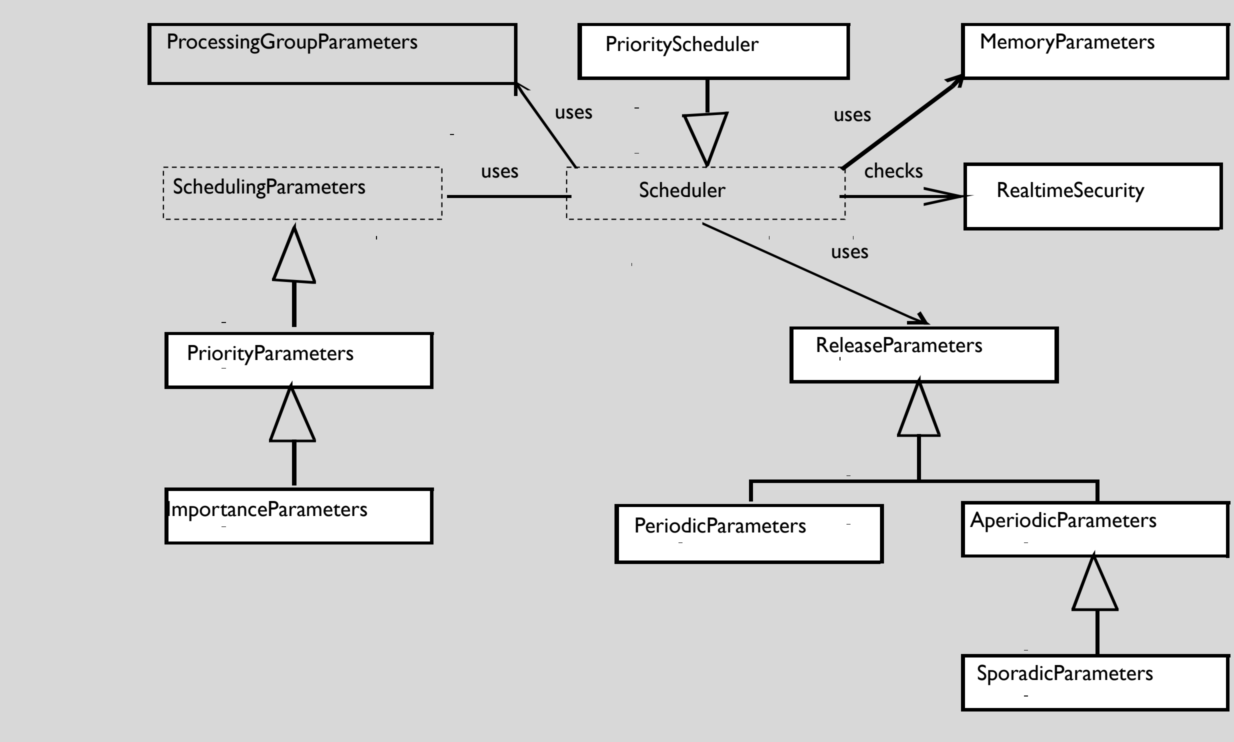
Scheduling Parameters

- Scheduling parameters are used by a scheduler to determine which object is currently the most eligible for execution
- The abstract class **SchedulingParameters** provides the root class from which a range of possible scheduling criteria can be expressed
  - The RTSJ defines only one criterion which is based on priority
  - High values for priority represent high execution eligibilities.

Schedulers

- Responsible for scheduling associated schedulable objects
- RTSJ supports priority-based scheduling via the **`PriorityScheduler`** (a fixed preemptive priority-based scheduling with 28 unique priority levels)
- `Scheduler` is an abstract class with `PriorityScheduler` a defined subclass
- An implementation could provide an EarliestDeadlineFirst scheduler
- Attempt by the application to set the scheduler for a thread has to be checked for the appropriate security permissions

Scheduling-related Classes



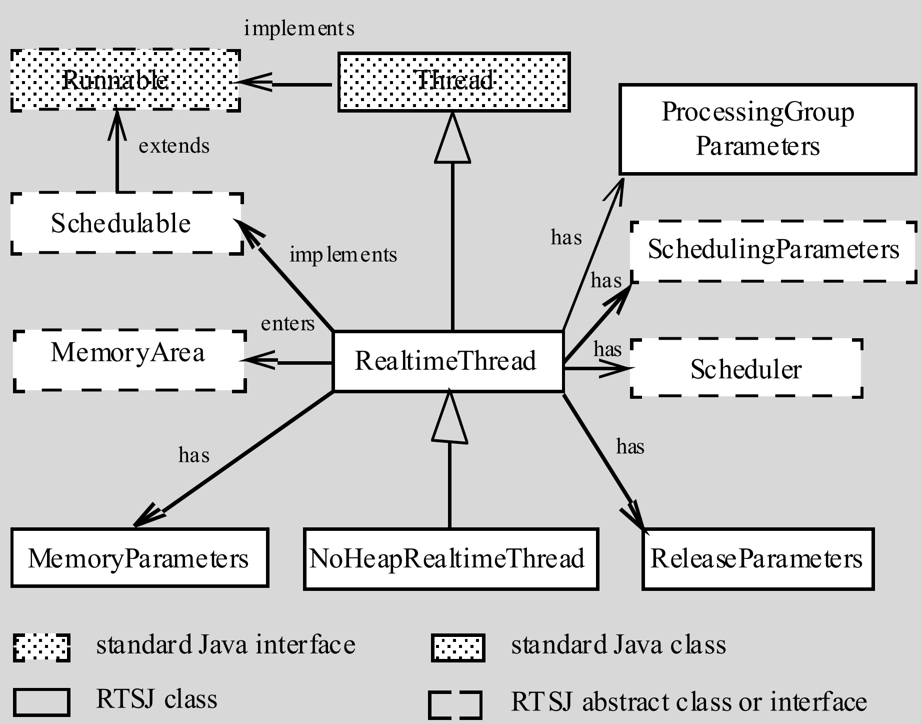
Meeting Deadlines

- A real-time system needs to
  - predict whether a set of application objects will meet their deadlines
  - report a missed deadline, a cost overrun, or min. inter-arrival violation
- For some systems it is possible to predict offline whether the application will meet its deadline
- For other systems, some form of on-line analysis is required
- The RTSJ provide the hooks for on-line analysis
- Irrespective of how prediction is done, it is necessary to report overruns
- The RTSJ provides an asynchronous event handling mechanism for this purpose

Real-Time Threads

- A schedulable object
- More than an extension of `java.lang.thread`
- No heap version ensure no access to the heap and therefore independence from garbage collection

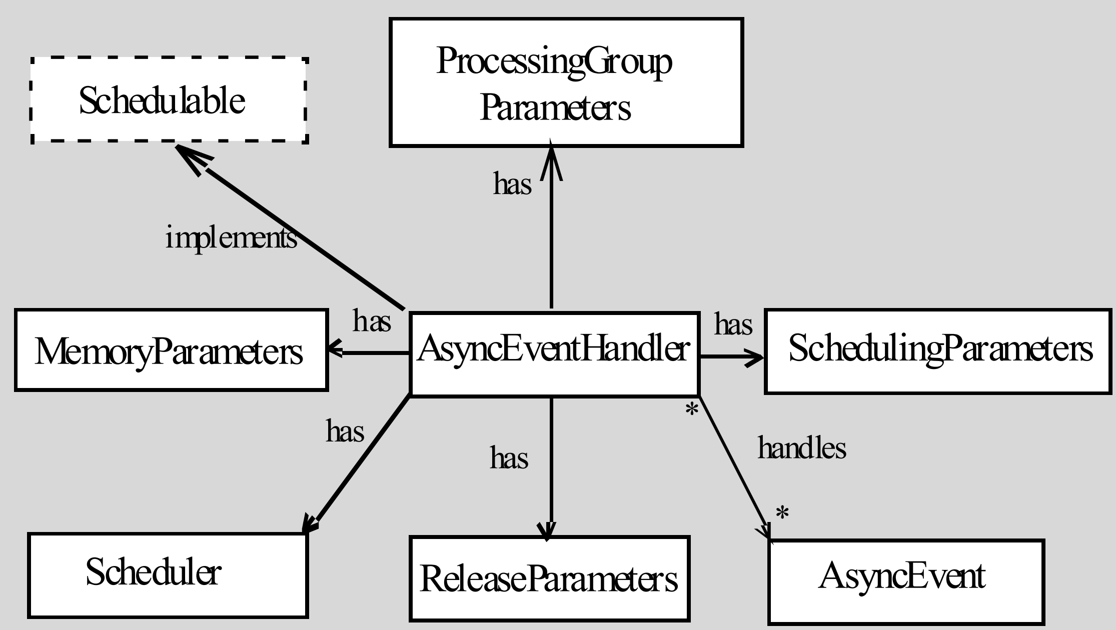
Real-time Threads



Asynchronous Event Handling

- Threads and RT threads are the abstractions to use to represent concurrent activities that have a significant life history. But it is also necessary to respond to events that happen asynchronously to a thread's activity
- Events may be happenings in the environment of an embedded system or notifications received from within the program
- It possible to have threads wait for them but this is inefficient
- From a RT perspective, events may require their handlers to respond within a deadline; hence, more control is needed over the order in which events are handled
- RTSJ generalizes event handlers to be schedulable entities

Async Events and their Handlers



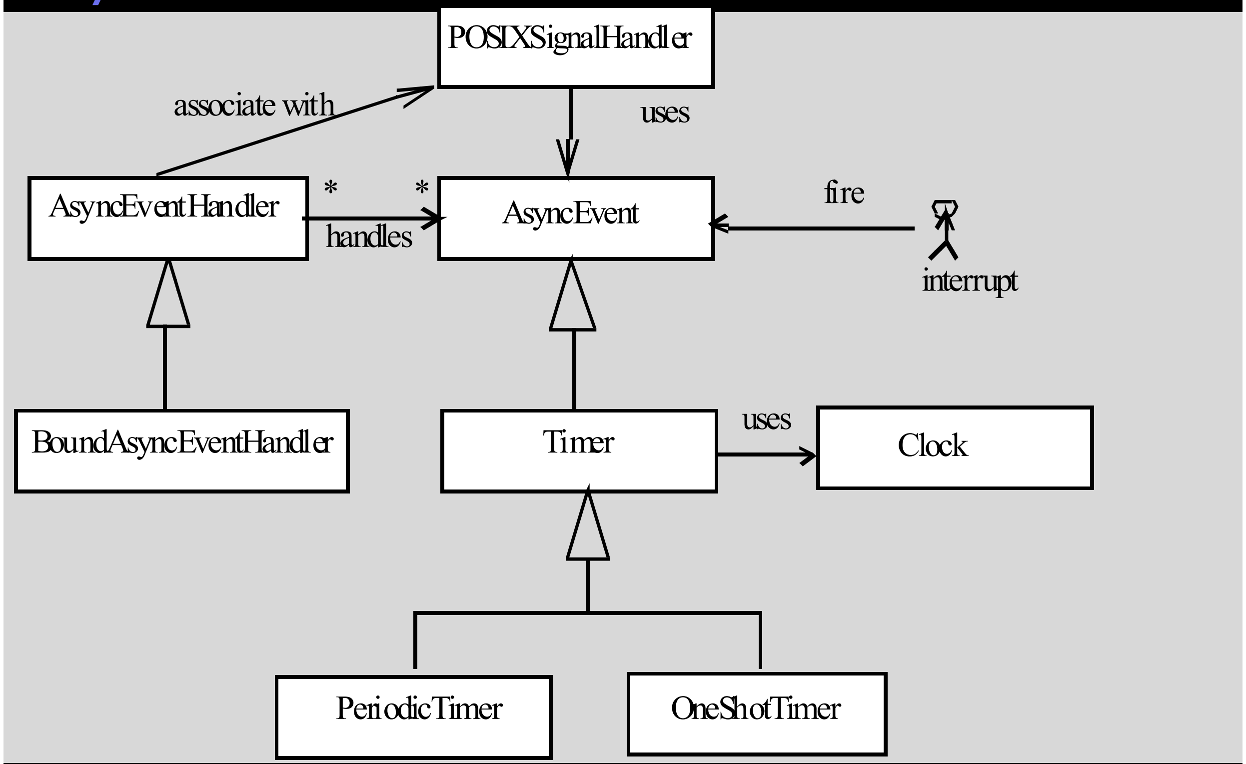
Handlers and Real-Time Threads

- In practice, an event handler will be associated dynamically with a RT thread when the handler is released for execution
- To avoid this overhead, it is possible to specify that the handler must be permanently bound to a RT thread
- Each **`AsyncEvent`** can have one or more handlers and the same handler can be associated with more than one event
- When the event occurs, all the handlers associated with the event are released for execution according to their `SchedulingParameters`

More on Async Events

- Asynchronous events can be associated with interrupts or POSIX signals or they can be linked to a timer
- The timer will cause the event to fire at a specified time
- This can be a one shot firing or a periodic firing

AsynEvent related Classses



Asynchronous Transfer of Control

- Asynchronous events allow the program to respond in a timely fashion to a condition
- They do **not** allow a particular schedulable object to be directly informed
- In many apps, the only form of asynchronous transfer of control that a real-time thread needs is a request for it to terminate itself
- Consequently, languages and OSs provide a kill or abort facility
- For RT systems, this is too heavy handed; instead what is required if for the schedulable object to stop what it is doing and to execute an alternative algorithm

ATC I

- In standard Java, it is the interrupt mechanism which attempts to provide a form of asynchronous transfer of control
- The mechanism does not support timely response to the "interrupt"
- Instead, a running thread has to poll for notification
- This delay is deemed unacceptable for real-time systems
- For these reasons, the RTSJ provides an alternative approach for interrupting a schedulable object, using asynchronous transfer of control (ATC)

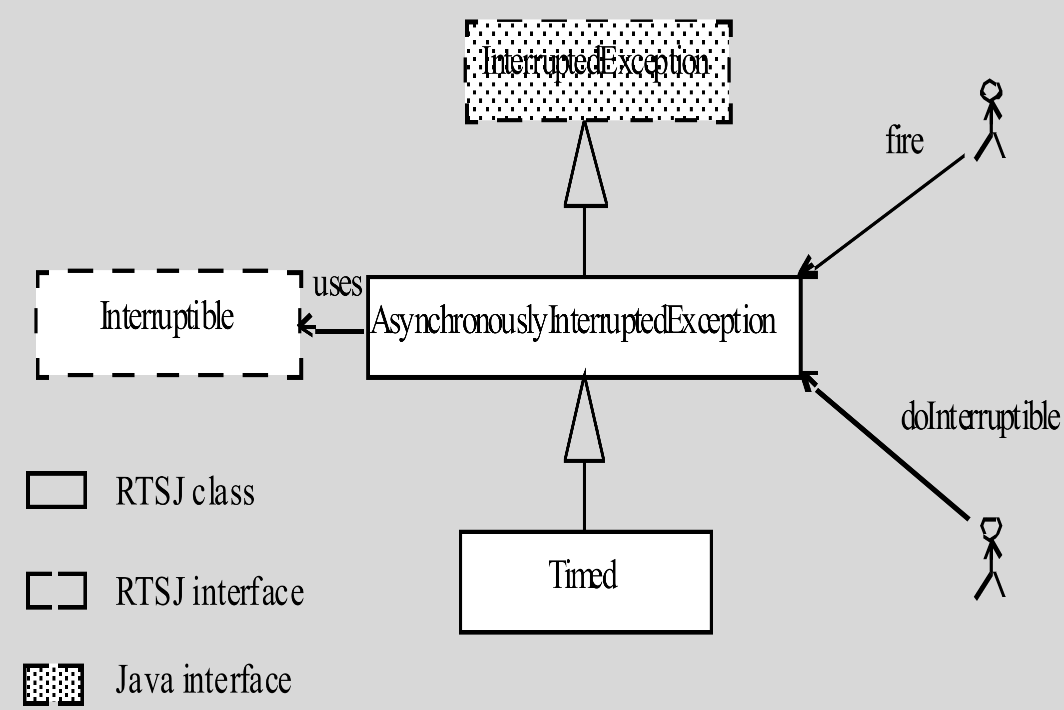
ATC II

- The ATC model is based on the following principles
  - A schedulable object must explicitly indicate that it is prepared to allow an ATC to be delivered
  - By default, schedulable object will have ATCs deferred
  - The execution of synchronized methods and statements always defers the delivery of an ATC
  - An ATC is a non-returnable transfer of control

ATC III

- The RTSJ ATC model is integrated with the Java exception handling facility
- An `AsynchronouslyInterruptedException` (AIE) class defines the ATC event
- A method that is prepared to allow an AIE indicate so via a throws `AsynchronouslyInterruptedException` in its declaration
- The `Interruptible` interface provides the link between the AIE class and the object executing an interruptible method

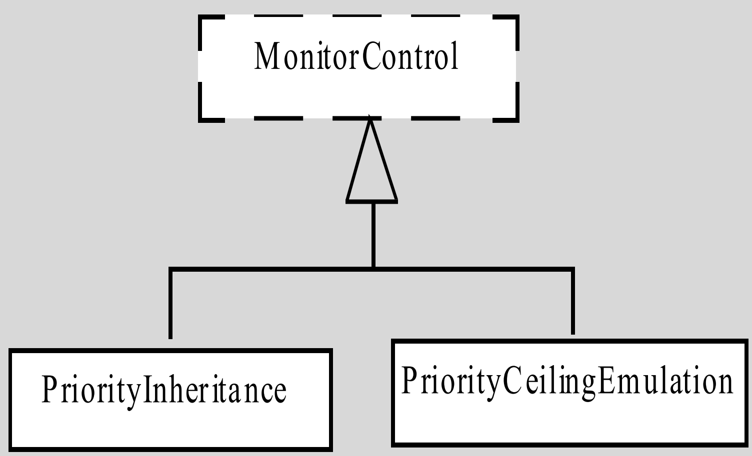
The ATC Classes and Interface



Synchronization

- Key to predicting the behavior of concurrent programs is understanding how threads communicate and synchronize
- Java provides mutually exclusive access to shared data via a monitor-like construct
- All synchronization mechanisms which are based on mutual exclusion suffer from priority inversion
- The problem of priority inversion is well-researched
- There are many priority inheritance algorithms; RTSJ support two: simple priority inheritance and priority ceiling emulation inheritance

RTSJ Classes for Priority Inheritance



Priority Inheritance and GC

- If RT threads want to communicate with plain threads then interaction with GC must be considered
- It is necessary to try to avoid the situation where a plain thread has entered into a mutual exclusion zone shared with a RTthread
- The actions of the plain thread results in garbage collection
- The RT thread then preempts GC but is unable to enter the mutual exclusion zone
- It must now wait for GC to finish and the plain thread to leave

Wait Free Communication

- One way of avoiding unpredictable interactions with GC is to provide a non-blocking communication mechanism for use between plain and RT threads
- RTSJ provides three wait-free non blocking classes:
  - `WaitFreeWriteQueue`: a bounded buffer; the read operation is synchronized; the write operation is not synchronized
  - `WaitFreeReadQueue`: a bounded buffer; the write operation on the buffer is synchronized; the read operation is not; the reader can request to be notified (via an asynchronous event) when data arrives
  - `WaitFreeDequeue`: a bounded buffer which allows both blocking and non-blocking read and write operations

Physical and Raw Memory Classes

- Allow objects to be placed into parts of memory with particular properties or access requirements; eg DMA memory, shared memory
  - Extensions of `MemoryArea` are the physical memory counterparts to the linear-time, variable-time and immortal memory classes
- Allow the programmer to access raw memory locations that are being used to interface to the outside world; e.g memory-mapped I/O device registers
  - Classes which can access raw memory in terms of reading and writing Java variables or arrays of primitive data types (int, long, float etc.)

Physical and Raw Memory Classes II