Asynchronous Event Handling and Safety Critical Java

Andy Wellings* and Minseong Kim

* Member of JSR 302

Structure

- Threads or event handling
- Why JSR 302 decided to use event handlers
- The JSR 302 concurrency model
- Known inconsistencies in the model
- Revised model
- Conclusions

Concurrency Models: Threads

- Support standardised across OSs
- Supported in most real-time languages
- Well established and problems well understood
  - real-time scheduling
  - priority inversion control
  - deadlocks (if use locks)
  - composability and scalability

Concurrency Models: Events and their Handlers

- More light-weight than threads
- Typically handlers are executed by one or more implementation-defined server threads
- Communication between handlers can be more straight forward if handler-server mapping known
- Real-time scheduling is more difficult

RTSJ

- Supports both real-time threads and asynchronous event handlers
- Version 1.1 has consistent support for periodic, aperiodic and sporadic activities using either approaches

SCJ Design Goals

- To define a subset of Java augmented with the RTSJ to support safety-critical systems development
- To support a programming model that is sufficiently limited to enable certification of applications using standards such as DO-178B Level A

Safety-critical Java

- Safety critical software varies considerably in complexity from application to application
  - At one end of the spectrum, the application consists of a single thread executing a single function on a single processor with a simple timing constraint
  - At the other end, the application is multi-threaded executing in multiple modes on multiple processors
- The RTSJ computation model is too rich and expensive for most safety critical systems
  - remove redundant features

Threads or Events?

- RT threads do not have an easily identifiable section of code that represents the work to be done on each release
  - It is the area of code inside a loop that is delimited by a call to the *waitForNextPeriod* or *waitForNextRelease* methods
- In contrast, an event handler has the *handleAsyncEvent* method which exactly contains this code
  - Hence static analysis tools are more easily facilitated
- A bound asynchronous event handlers is equivalent to a real-time thread in functionality and its impact of scheduling
  - little is lost by its use over that of real-time threads

The Mission Concept

- A mission consists of a bounded set of limited schedulable objects
- For each mission, a specific block of memory is defined called mission memory
  - Objects created in mission memory persist until the mission is terminated, and their resources will not be reclaimed until the mission is terminated
- A mission starts in an initialization phase during which objects may be allocated in mission memory and immortal memory by an application
  - There is no garbage-collected heap

Missions continued

- All schedulable objects are created during the initialization phase
  - When a mission's initialization has completed, its execution phase is entered, and all the created schedulable objects are started
  - During the execution phase, no new schedulable objects can be created
- When a schedulable object is started, its initial memory area is a scoped memory area that is entered when the schedulable object is released, and is exited (i.e., emptied) when the schedulable object completes that release
  - This scoped memory area is not shared with other schedulable objects. Hence SCJ has simplified many of the complexities that are inherent in the full RTSJ memory management model

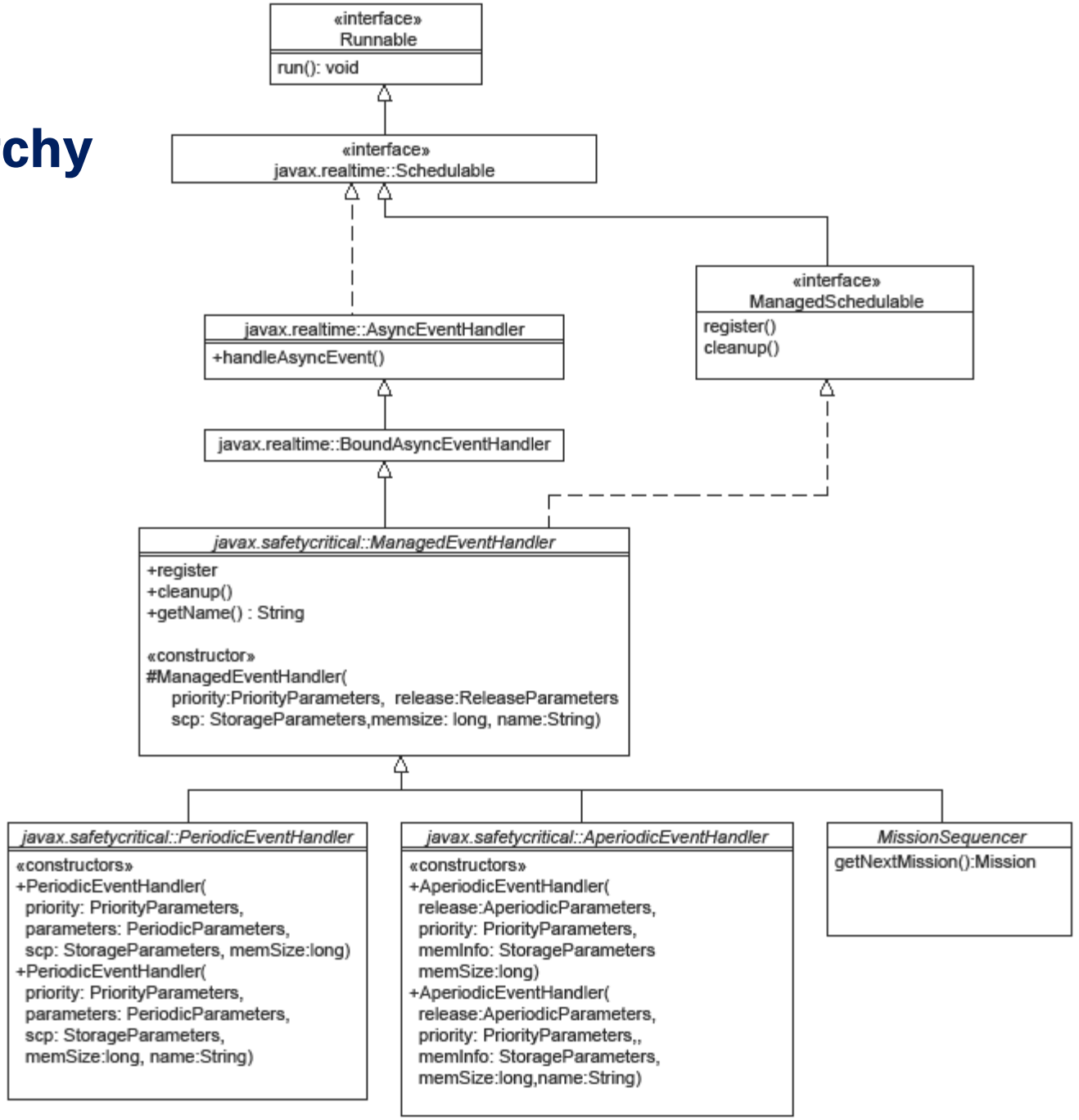
Missions and Handlers

- All handlers are managed by the enclosing mission hence use of the RTSJ classes themselves is prohibited
- There is no support, for example, for:
  - on-line feasibility analysis
  - sporadic release parameters
  - dynamic priorities
  - cost monitoring or enforcement
  - dynamic binding between events and their handlers
  - manipulation of fireCount
- The restricted programming model is enforced by the removal of methods (and constructors) and the provision of new classes and a new interface to support mission management

Managed Schedulable Objects

- Objects that are mission-aware and therefore register themselves with a mission manager when they are created
- They also provide cleanup code that can be invoked by the manager when the mission terminates
- The ManagedEventHandler abstract class is an RTSJ bound asynchronous event handler that is mission aware
- SCJ supports periodic and aperiodic versions of this class
- The new classes are defined in the javax.safetycritical package and are fully implementable using standard RTSJ

The SJC Event Handling Hierarchy



Compliance Levels for SCJ Programs

- Level 0
  - Single mission sequencer, essentially a cyclic executive
  - Main programming abstraction: non-self suspending periodic event handlers
- Level 1
  - Single mission sequencer, essentially fixed priority scheduling
  - Main programming abstraction: non-self suspending periodic and aperiodic event handlers
- Level 2
  - Nested mission sequencers, essentially fixed priority scheduling
  - Main programming abstraction: periodic and aperiodic event handlers and simple real-time threads – can self suspend but not while holding nested locks

Inconsistencies in SCJ

- With RTSJ
  - ASEH are mapped to server threads
  - Most implementations do a N handlers to 1 thread mapping, with late binding
  - BASEH has a 1 to 1 Mapping
- With SCJ
  - All handlers are BASEH
  - At level 0 there is effectively a N to 1 Mapping
  - At Levels 1 and 2 it is 1 to 1

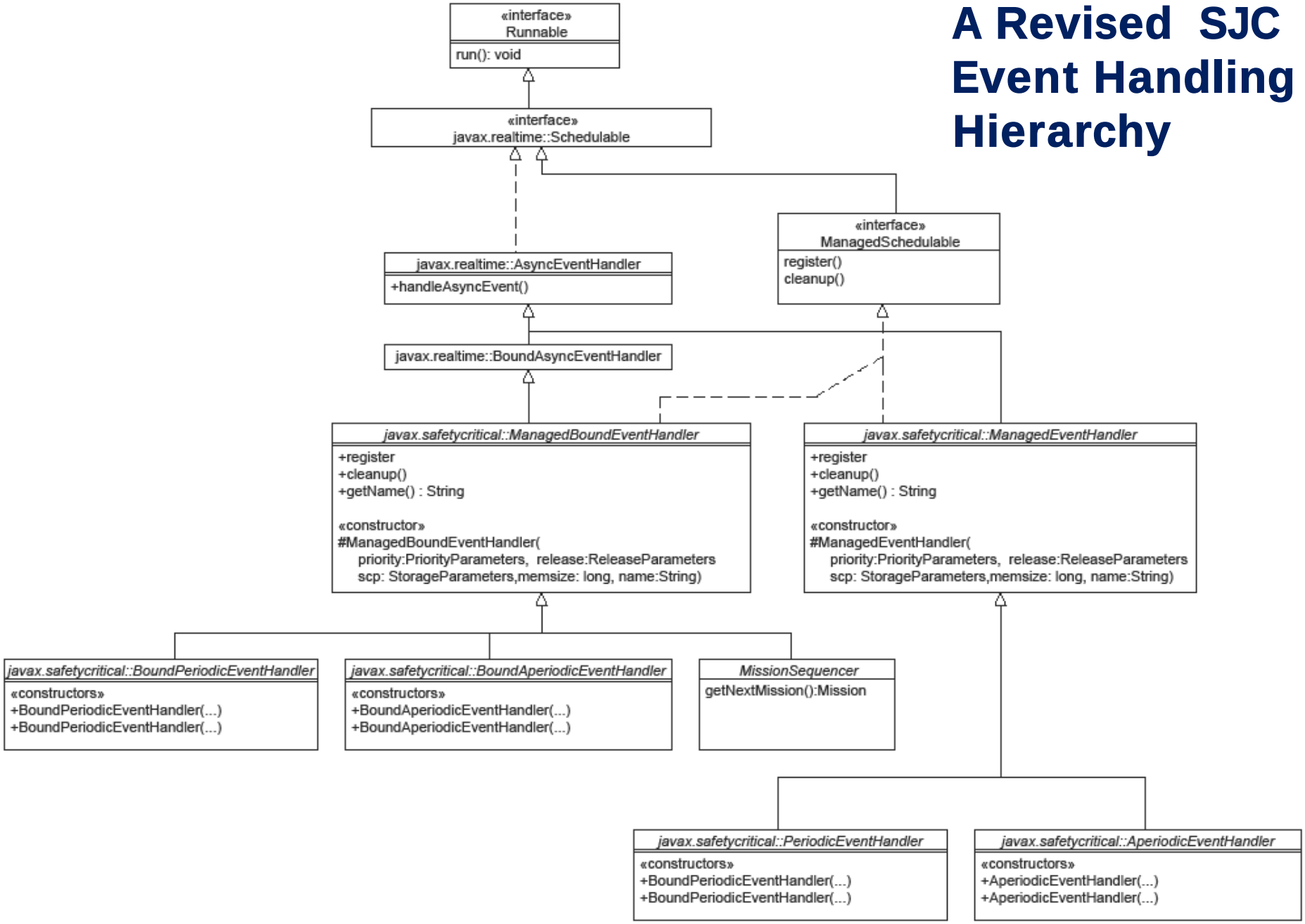
Observations

- If ASEH are non-self suspending then it is sufficient to have one server thread per priority level (per machine) to support any number of handlers.
- If ASEH can potentially-suspend, in the worst case, you need one thread per handler
- RTSJ does not restrict handlers, SCJ does

A more consistent SCJ Model

- ASEH are non self suspending
- BASEH are potentially self suspending
- Level 0 and Level 1 handlers should be based on ASEH
- Level 2 can choose between ASEH and BASEH

A Revised SJC Event Handling Hierarchy



Summary

- A level 0 application can only use the PeriodicEventHandler class
  - As this is an asynchronous event handler that is defined to be non self-suspending, a single server thread can be used.
- A level 1 application can only use the PeriodicEventHandler and the AperiodicEventHandler classes.
  - Again, as these are non self-suspending asynchronous event handlers, server technology can be used
  - It is up to the implementation to decide how best to map the handler to the underlying threading model
  - This approach must be documented to facilitate timing analysis
  - Note that a single server thread allocated to each handler is still a valid implementation approach with this model

Summary continued

- A Level 2 application can use all types of event handlers as long as the program conforms to the implied constraints
  - As the handler type is clearly identified, static analysis tools can easily determined if potentially self-suspending operations are being called
  - The implementation is free to optimize the support for non self-suspending handlers

Unfortunately

- Java does not support multiple inheritance and as a consequence the support for managed events handlers has to be replicated
- It is this replication that is ugly and one of the reason why only bound asynchronous event handlers are used in the current SCJ
- Another reason is the increase in complexity of the run-time environment to support the mapping of event handlers to threads
- However, we note that on a single processor this is a static mapping determined by the handlers priority