Programming Language Considerations for Real-Time Computation

What are likely requirements?

- Clocks/timers

RT Extensions to POSIX Spec

- Toward **unifying** the real-time application and OS space, the Real-Time Portable Operating System based-on Unix (RT-POSIX 1003.1 b) ISO standard was created.
- POSIX is a standard that has been very successful and widely adopted in a wide-range of OS implementations, for both mission-critical and general-purpose computing.
- RT-POSIX defines an extension to POSIX intended to address the needs of hard and soft real-time systems.

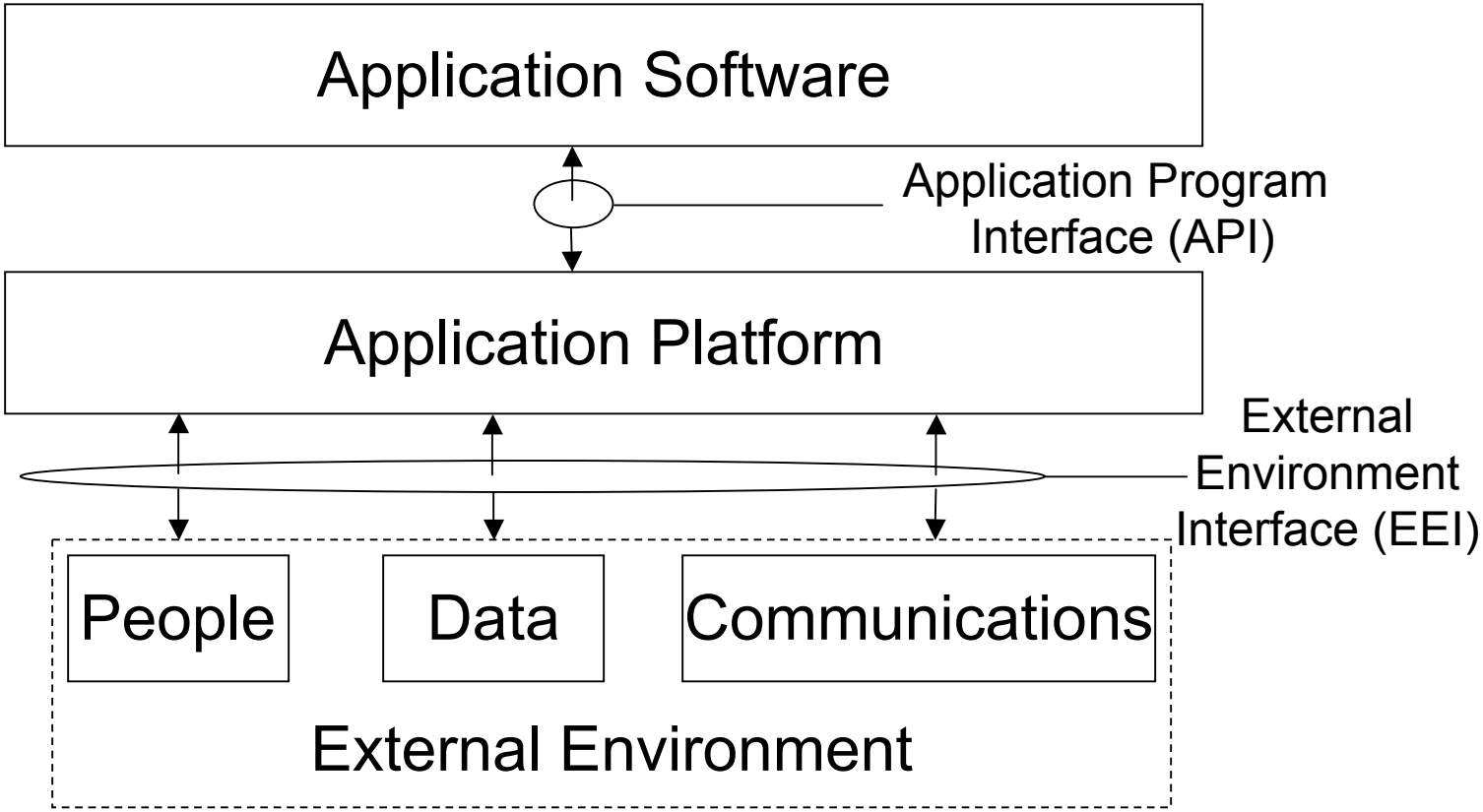
Source: Bruno and Bollella, Real-Time Java Programming

POSIX Interfaces

- There are also two types of standard specified in the POSIX model
  - The application program interface (API)
  - The external environment interface (EEI)

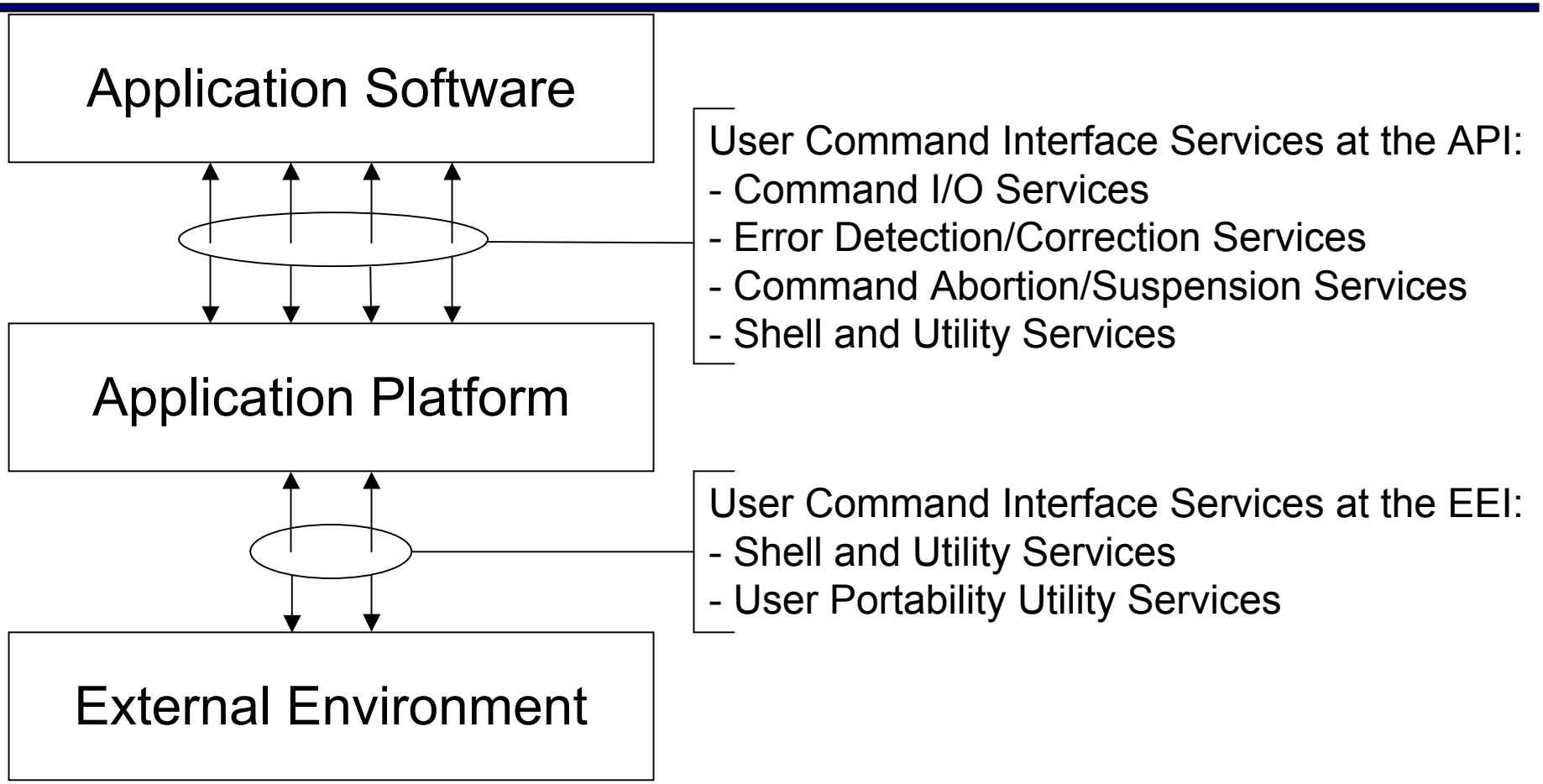
Source: ITSW 2436/Kenneth R. Frazer

POSIX Interfaces



Source: ITSW 2436/Kenneth R. Frazer

POSIX User Commands



Source: ITSW 2436/Kenneth R. Frazer

- POSIX standards represent approximately $80 billion of the $250+ billion Unix market over the last decade
  - They also provide the basis for significant growth in the future
- Most system vendors are now conforming to POSIX standards (specifically IEEE 1003.1)
  - Even Microsoft provides a set of POSIX utilities with the Windows NT 4.0 Resource Kit
- POSIX is no longer a market differentiator; it is expected that a Unix product comply

Source: ITSW 2436/Kenneth R. Frazer

RT-POSIX Requirements

- **Preemption**: True task preemption must be supported using task priority rules that are strictly obeyed.
- **Priority Inversion Control**: Although it cannot guard against deadlocks, **priority inheritance** ensures that lower-priority threads will never be able to block higher-priority threads due to classic priority inversion.
- **Periodic, Aperiodic, and Sporadic Threads**: the POSIX standard requires only processes be implemented. For real-time applications, threads of different priority may need to run to perform a task.

RT-POSIX Requirements, cont'd

- **High-Resolution Timers**: The RT-POSIX standard states that up to 32 timers per process must be supported, and that timer overruns (when a timer goes beyond its chosen duration) be recorded.
- **Schedulers**
- **Scheduled Interrupt Handling**
- **Synchronous and asynchronous I/O**
- and more

Java Threads for Real-Time?

- In original Java, every thread has a priority.
- When there is competition for processing resources, threads with higher priority are **generally** executed in preference to threads with lower priority.
- **But**, such preference is **not a guarantee** that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

Java-POSIX Compliance Gap

- Strict thread priority control
- Thread preemption based on thread priority
- Thread priority inversion control
- Resource locking with minimal wait states (limited critical sections)
- Low-pause garbage collector
- Synchronous and asynchronous event processing
- Access to physical memory
- Access to hardware interrupts, and the ability to schedule them
- The ability to lock code and the heap into memory

Extending Java for Real-Time

- Disclaimer:
  - We will not argue whether or not Java can be the basis for the ideal, or even a good, real-time language.
  - The exercise of attempting to make it so is instructive, in terms of the many language implementation issues that have been considered.

Demos

- SlotCar: http://www.youtube.com/watch?v=Qfhkp4wid2I
- Inverted Pendulum demo: http://www.youtube.com/watch?v=lXct7Bzdhzw
- PLC replacement http://www.youtube.com/watch?v=g5Ky2ApC4Y4

JSRT: Java Specification for Real-Time Criteria

Java Environment Agnostic: the RTSJ shall not preclude its implementation in any known Java environment or version of the development kit, such as the Java SE or Java ME platforms. Instead, the goal is for it to allow general-purpose Java implementations for use in server and/or embedded environments.

Predictability: predictable Java application execution shall be the primary concern, with tradeoffs in the area of general purpose computing being made where necessary.

Compatibility: existing Java application code shall run properly on an implementation of the RTSJ.

Java Syntax: there shall be no additions or changes in terms of Java language keywords or syntax.

WORA: the RTSJ respects the importance of the write-once-run-anywhere (WORA) mantra for Java, but shall favor predictability over WORA where necessary.

Consideration for Current and Future Work: although the RTSJ addresses current real-time systems practice, provisions are made to allow it to be extended for future work in this area.

Flexible Implementation: the RTSJ shall allow for flexibility in decisions on implementation. Such decisions include performance/footprint tradeoffs, the choice of scheduling algorithms, and so on.

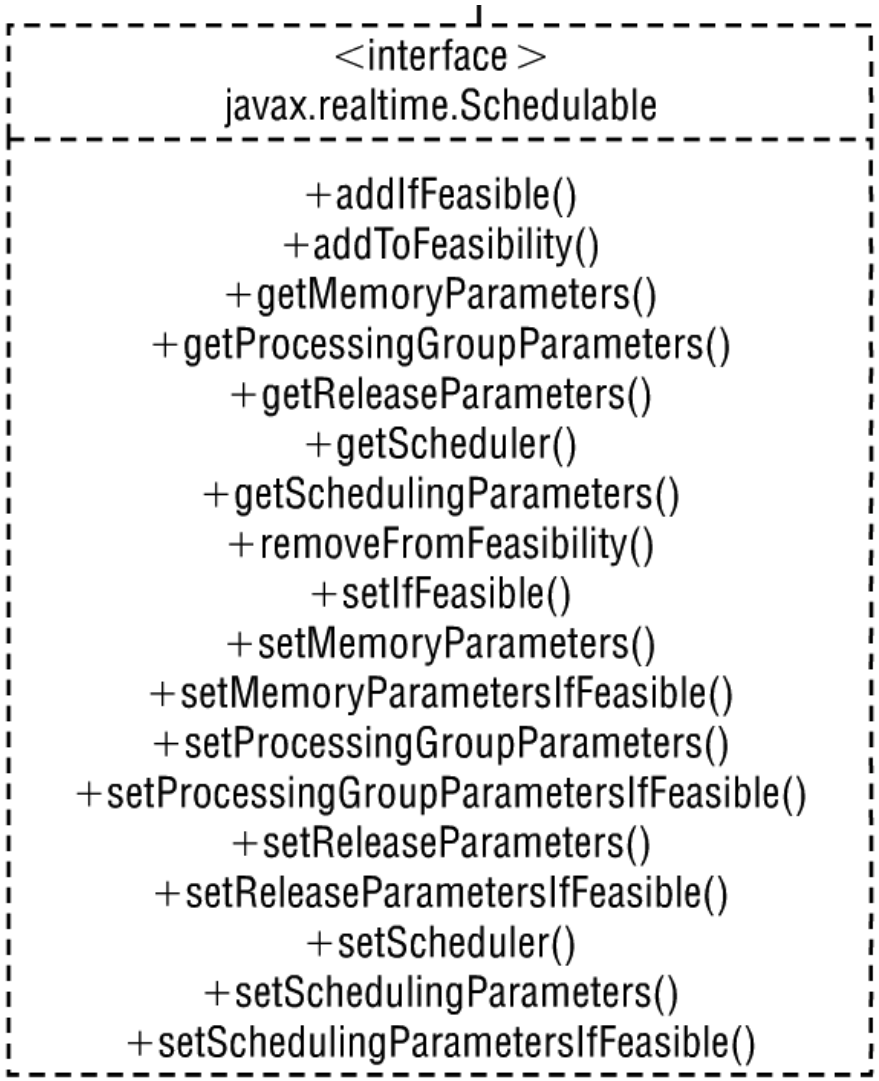
Java Real-Time Implementation

- Don’t want to add to the language.
- Rather add new libraries.
- But, unlikely that a traditional JVM (Java Virtual Machine) will suffice for the underlying implementation.
- RT JVM may also rely on OS features for implementation support.

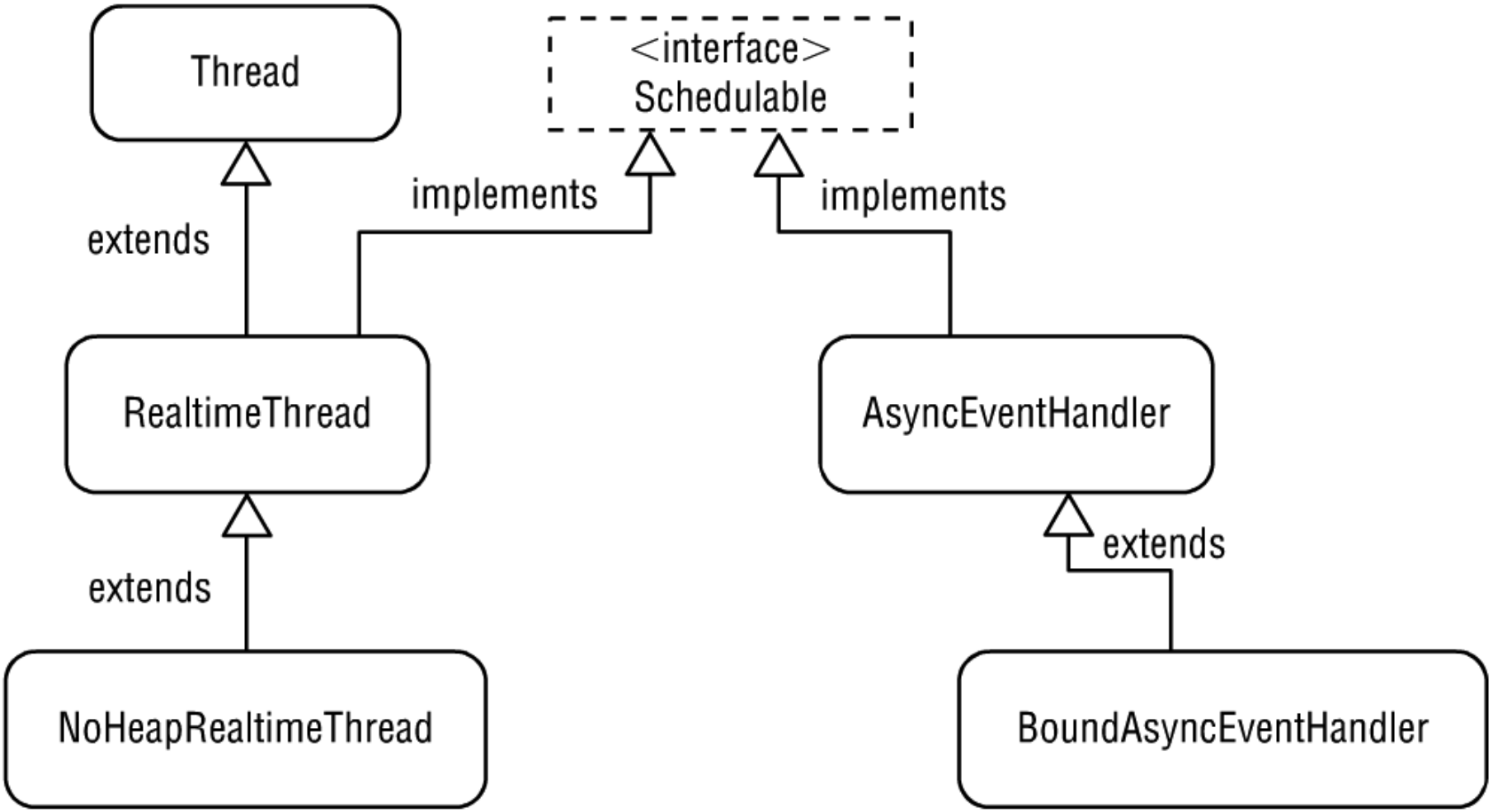
Schedulable interface

- **Schedulable** is used to classify things that can be scheduled by a **Scheduler**.
- Schedulable extends **Runnable**.
- Lots of methods are required to be an implementer.
- Two implementing classes are:
  - RealTimeThread
  - AsynchEventHandler

javax.realtime.Schedulable interface extends Runnable



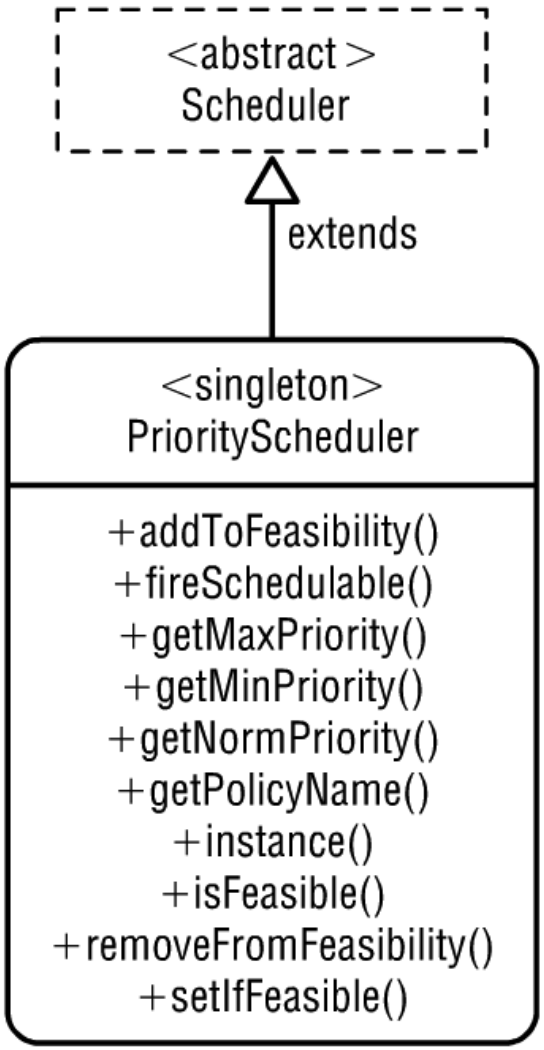
Implementers of Schedulable



Priority- vs. Time-Share Scheduling

- Code running within a **Schedulable** object, such as RealtimeThread, executes according to a **fixed-priority, preemptive** scheduling policy.
- Threads scheduled according to the above policy are granted the **highest range of scheduling priorities** on the system, and preempt all threads running at lower priority.
- Threads running at a **regular** Java priority level (JLTs) are scheduled according to a **time-sharing policy**, which provides **fair allocation** of processor resources among threads.

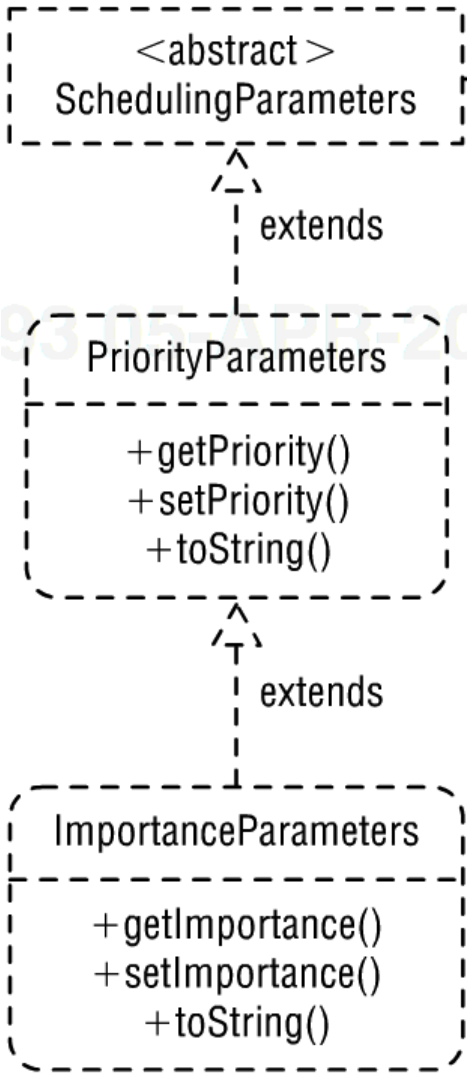
Scheduler abstract class and the most common extension



Singleton PriorityScheduler can

- be obtained through its static instance method.
- determine the various priority ranges of the given scheduler, the priority of a given thread, and the feasibility of the current set of Schedulable objects.
- alter the running system by adding new Schedulable objects with a feasibility check.
- change an existing Schedulable's parameters and test for feasibility.
- trigger an asynchronous event to fire.

Scheduling Parameters



ImportanceParameters

Importance is an additional scheduling metric that may be used by some priority-based scheduling algorithms **during overload conditions** to differentiate execution order among threads of the same priority.

In some real-time systems an external physical process determines the period of many threads. If **rate-monotonic priority** assignment is used to assign priorities many of the threads in the system may have the **same priority because their periods are the same**. However, it is conceivable that **some threads may be more important than others** and in an overload situation importance can help the scheduler decide which threads to execute first.

The RTSJ strongly suggests to implementers that a **fairly simple subclass** of PriorityScheduler that uses importance can offer value to some real-time applications.

Support for Periodic Scheduling

```
public class RealtimeThread extends java.lang.Thread
        implements Schedulable
{
  ...
  public boolean waitForNextPeriod()
          throws IllegalThreadStateException;

  public void deschedulePeriodic();

  public void schedulePeriodic();
  ...
}
```

Support for Periodic Scheduling

- The `ReleaseParameters` associated with a real-time thread can specify asynchronous event handlers which are scheduled by the system if the thread misses its deadline or overruns its cost allocation.

- For deadline miss, no action is immediately performed on the thread itself. It is up to the handlers to undertake recovery operations.

- If no handlers have been defined, a count is kept of the number of missed deadlines.

NoHeapRealtimeThread ?

- In standard Java, threads can be arbitrarily delayed unpredictably due to garbage collection.
- The RTSJ has attacked this problem by allowing objects to be created in memory areas other than the heap. These areas are not subject to garbage collection.
- A no-heap real-time thread is a real-time thread which never accesses heap memory areas.
- Consequently, it can safely be executed even when the garbage collection is occurring.

NoHeapRealtimeThread ?

- The constructors for the `NoHeapRealtimeThread` class require reference to a memory area, within which all memory allocation by this thread will be performed.

- An unchecked exception is thrown if the `HeapMemory` area is passed.

class javax.realtime.MemoryArea

Constructor Summary	
protected	**MemoryArea**(long sizeInBytes)
protected	**MemoryArea**(long sizeInBytes, java.lang.Runnable logic)
protected	**MemoryArea**(SizeEstimator size)
protected	**MemoryArea**(SizeEstimator size, java.lang.Runnable logic)

class javax.realtime.MemoryArea

Method Summary	
void	**enter**() Associate this memory area to the current real-time thread for the duration of the execution of the `run()` method of the `java.lang.Runnable` passed at construction time.
void	**enter**(java.lang.Runnable logic) Associate this memory area to the current real-time thread for the duration of the execution of the `run()` method of the given `java.lang.Runnable`.
void	**executeInArea**(java.lang.Runnable logic) Execute the run method from the `logic` parameter using this memory area as the current allocation context.
static MemoryArea	**getMemoryArea**(java.lang.Object object) Returns the `MemoryArea` in which the given object is located.
long	**memoryConsumed**() An exact count, in bytes, of the all of the memory currently used by the system for the allocated objects.
long	**memoryRemaining**() An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.
java.lang.Object	**newArray**(java.lang.Class type, int number) Allocate an array of T in this memory area.
java.lang.Object	**newInstance**(java.lang.Class type) Allocate an object in this memory area.
java.lang.Object	**newInstance**(java.lang.reflect.Constructor c, java.lang.Object[] args) Allocate an object in this memory area.
long	**size**() Query the size of the memory area.

javax.realtime.ImmortalMemory

- singleton class
- implements javax.realtime.MemoryArea
- “unexceptionally available to all schedulable objects and Java threads for use and allocation”

javax.realtime.ScopedMemory

- abstract base class
- implements javax.realtime.MemoryArea
- “unexceptionally available to all schedulable objects and Java threads for use and allocation”

javax.realtime.ScopedMemory

The **enter**() method of ScopedMemory is one mechanism used to make a memory area the current allocation context.

Entry into the scope is accomplished, for example, by calling the method:

```
public void enter(Runnable logic)
```

where logic is a instance of Runnable whose run() method represents the entry point of the code that will run in the new scope.

The other mechanism for activating a memory area is making it the initial memory area for a real-time thread or async event handler.

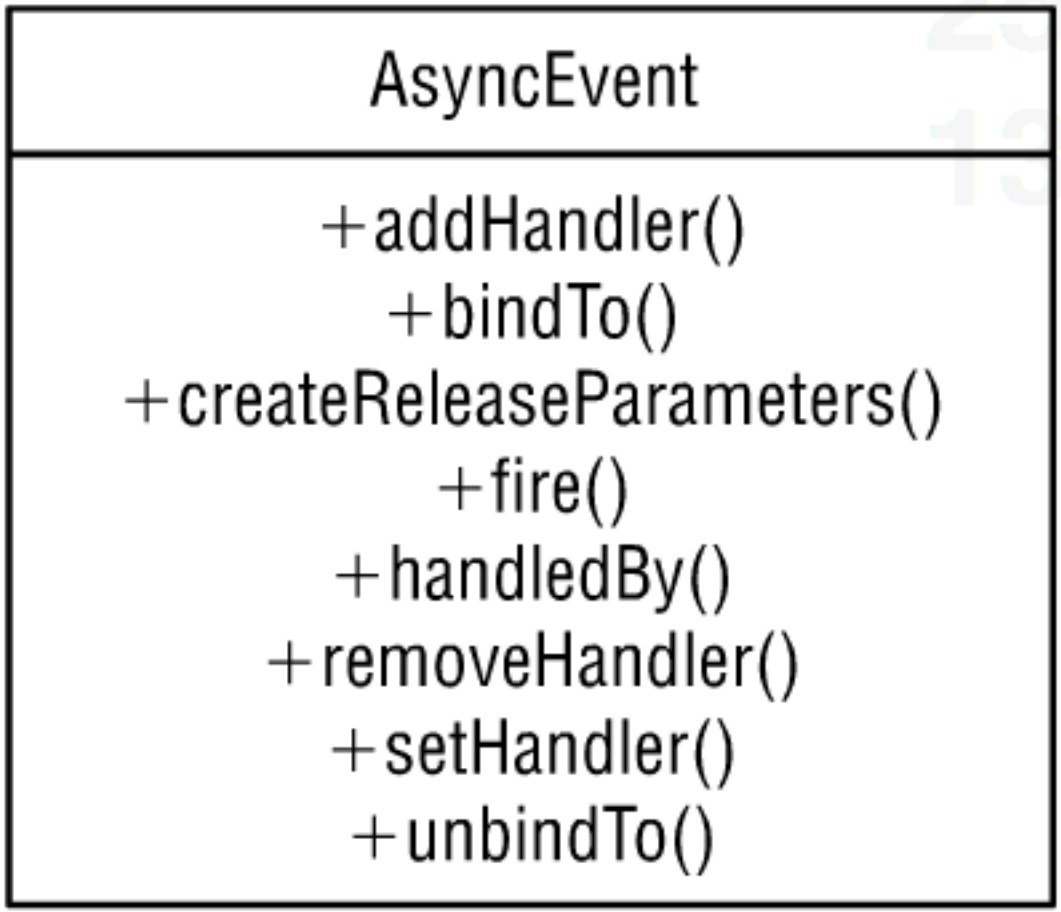
Asynchronous Events

- Analogous to POSIX signals
- Can be external or internal
- Can register handlers with events

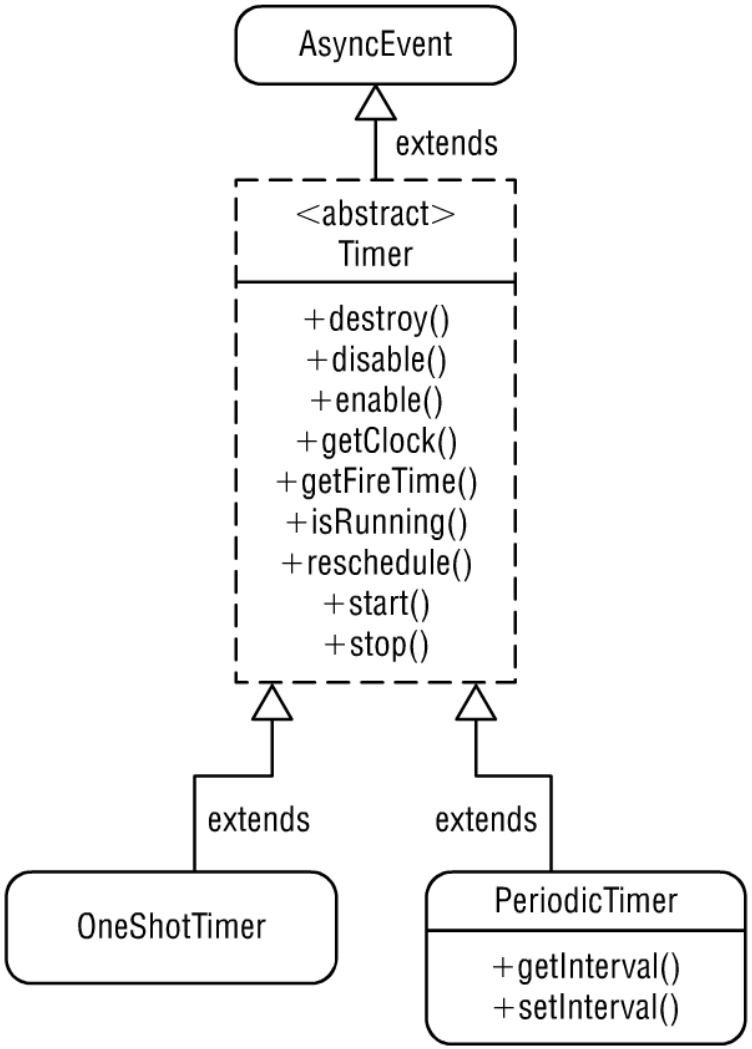
Two Event-Related Classes

- **AsyncEvent** —an object of this type represents the event itself. As of the RTSJ, the object **does not contain related event data**. This needs to be delivered through other means.
- **AsyncEventHandler** —an RTSJ Schedulable object that is executed by the AEH facility within the real-time Java VM when the related event is fired.
  - This object has associated ReleaseParameters , SchedulingParameters, and MemoryParameters objects, that indicate how the event handler is to be scheduled.
- No need to create dedicated threads for handling events.

AsyncEvent Methods



Example: Timer-Based Events



OneShotTimer

- A timed AsyncEvent driven by a clock.
- Will fire once, when the clock time reaches the timeout time.
- If the clock time has already passed the timeout time, it will fire immediately.

PeriodicTimer

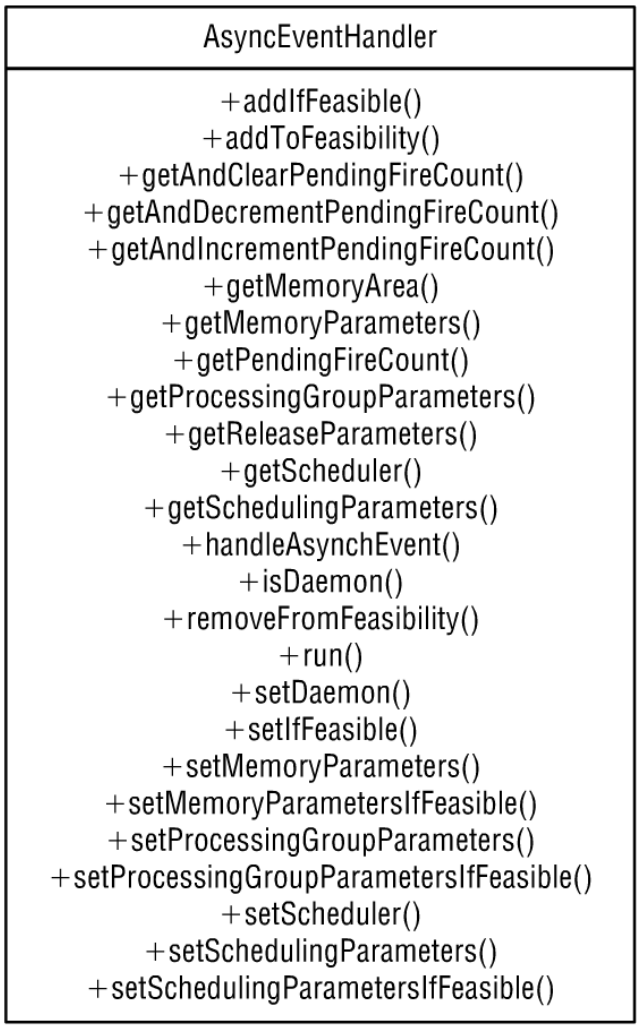
- An AsyncEvent whose **fire method** is executed periodically.
- Periods:
  - If a **clock** is given, calculation of the period uses the increments of the clock.
  - If an **interval** is given,the system guarantees that the fire method will execute interval time units after the last execution or its given start time as appropriate.
- If disabled, still counts, and if enabled at some later time, it will fire at its next scheduled fire time.
- Similar to a thread with PeriodicParameters except that it is lighter weight.

javax.realtime.Clock

(extends Object)

Method Summary	
static Clock	**getRealtimeClock**() There is always one clock object available: a realtime clock that advances in sync with the external world> This is the default Clock.
abstract RelativeTime	**getResolution**() Return the resolution of the clock -- the interval between ticks.
AbsoluteTime	**getTime**() Return the current time in a freshly allocated object.
abstract void	**getTime**(AbsoluteTime time) Return the current time in an existing object.
abstract void	**setResolution**(RelativeTime resolution) Set the resolution of this.

EventHandler Methods



Interfacing with POSIX signals

```
import javax.realtime.*;
public class MyApp {
    SigintHandler handler = null;
    final int RTSJ_MAX_PRI =
        PriorityScheduler.instance().getMaxPriority();

    class SigintHandler extends AsyncEventHandler {
        public SigintHandler() {
            setSchedulingParameters(
                new PriorityParameters(RTSJ_MAX_PRI));
        }

        public void handleAsynchEvent() {
            System.out.println("SIGINT occurred");
        }
    }

    public MyApp() {
        handler = new SigintHandler();
        POSIXSignalHandler.addHandler(
                POSIXSignalHandler.SIGINT, handler);
    }
```

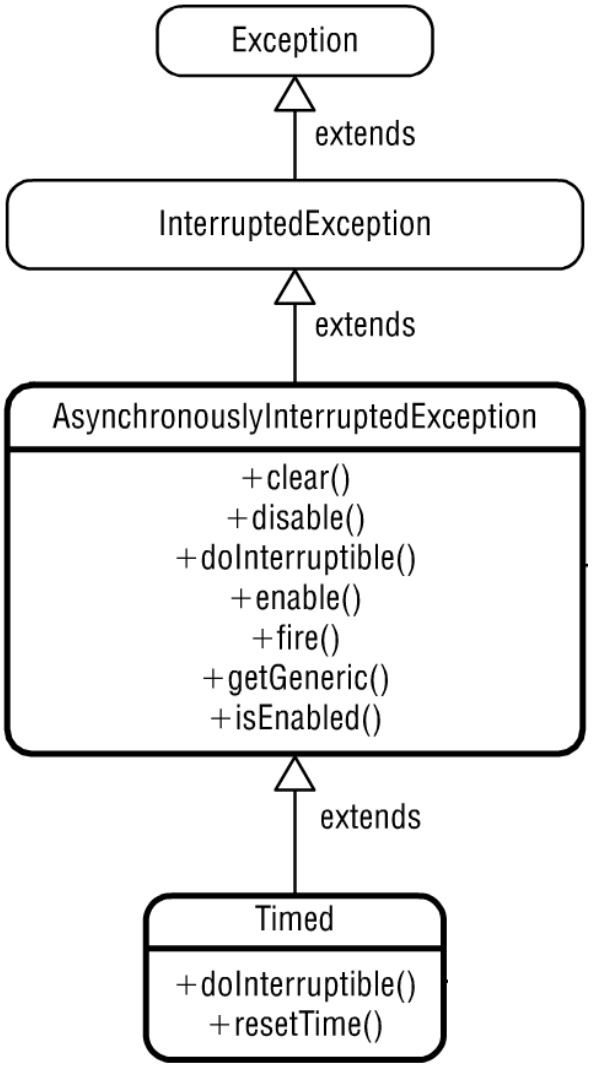
POSIX Signal Interfacing

- Java RTS spawns an internal thread to handle POSIX signals, and notify your application when an event of interest fires.
- By default, this thread runs at the RTSJ MaxPriority value. events in a timely manner.

ATC: Asynchronous Transfer of Control

- Transfer control between RT threads using Interrupt and Exception Mechanisms.

AsynchronouslyInterruptedException



Asynchronous Transfer of Control

- No interruption of synchronized methods/blocks.
- No automatic resumption at point of interruption.
- Target must have **explicit catch** clause for the **AsynchronouslyInterruptedException** (no “catch all”).
- Target must acknowledge interruption explicitly, by calling **doInterruptible**() or **AsynchronouslyInterruptedException**.clear().

class Timed

Creates a scope in a RealtimeThread for which **interrupt()** will be called at the expiration of a **timer**.

This timer will begin measuring time at some point between the time **doInterruptible()** is invoked and the time the **run()** method of the Interruptible object is invoked.

Each call of doInterruptible() on an instance of Timed will restart the timer for the amount of time given in the constructor, or the most recent invocation of resetTime().

All **memory use** of Timed occurs during construction or the first invocation of doInterruptible(). Subsequent invokes of doInterruptible() do not allocate memory.

class RealtimeSystem

- Provides a means for tuning the behavior of the implementation by specifying parameters such as
  - **maximum number of locks** that can be in use concurrently
  - **monitor control policy**

- Provides a mechanism for obtaining access to
  - the security manager
  - garbage collector
  - scheduler

  to make queries from them or to set parameters.

class RawMemoryAccess

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

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

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

Electronic Piano Example

```
class InputEvent {
  static final int KEY_PRESS = 1;
  static final int KEY_RELEASE = 2;
  static final int PEDAL_PRESS = 3;
  static final int PEDAL_RELEASE = 4;

  int type;

  /**  The key that was pressed                */
  int key;

  /** The time the key was pressed in
   *  milliseconds aligned with
   *  the real-time clock's absolute time..
   */
  long pressTime;

  /** The interval between sensor 1 and sensor 2 in microseconds. */
  int interval;
}
```

Electronic Piano Example

```
class InputHandler extends AsyncEventHandler
{
. . .
 public void handleAsyncEvent()
    {
    ToneEvent toneEvent = null;
    mux.getEvent(inputEvent);

    switch (inputEvent.type)
      {
    case InputEvent.KEY_PRESS:
      toneEvent = events.select(inputEvent.key);
      toneEvent.startTime.set(inputEvent.pressTime, 0);
      toneEvent.startTime.add(LATENCY, toneEvent.startTime);
      toneEvent.startTone(inputEvent.interval,
            toneEvent.startTime);
      break;

    case InputEvent.KEY_RELEASE:
    . . .
    }
  }

}
```