US Headquarters:

104 Fifth Avenue, 15th Floor
New York, NY 10011

+1-212-620-7300 (voice)
+1-212-807-0162 (FAX)

European Headquarters:

8 rue de Milan
75009 Paris France

+33-1-4970-6716 (voice)
+33-1-4970-0552 (FAX)

www.adacore.com

Real-Time Java™ for Ada Programmers

Ben Brosgol
brosgol@adacore.com

Tutorial– SIGAda 2005
Atlanta, GA

Monday, 21 November 2005

AdaCore

Tutorial outline

Part 1 - Basics

- Language requirements for real-time programming
- Java overview, thread model, and assessment for real-time applications
- Summary of underlying issues: priority inversion, garbage collection

Part 2 – Real-Time Specification for Java ("RTSJ")

- Overview and design goals
- Scheduling
- Memory Management

Part 3 – RTSJ (continued)

- Synchronization / priority inversion management
- Asynchronous event handling

Part 4 – RTSJ (concluded) and other real-time Java topics

- Asynchronous transfer of control
- Time and timers
- Low-level features
- Real-time Java in practice
- Conclusions

Preface

Audience background

- Some knowledge of (sequential) Java
- Some knowledge of real-time issues
- No knowledge of Java thread model
- No knowledge of the real-time Java efforts

Presenter background

- Primary member of “Expert Group” for original design of RTSJ
- Member of Technical Interpretation Committee for RTSJ
- Reviewer of J-Consortium Core Extensions
- Author of several Ada / Java comparison papers
- Frequent presenter at Ada Europe and SIGAda conferences
- Participant in Ada 83 and Ada 95 design efforts
- Member of AdaCore senior technical staff

Part 1: Basics

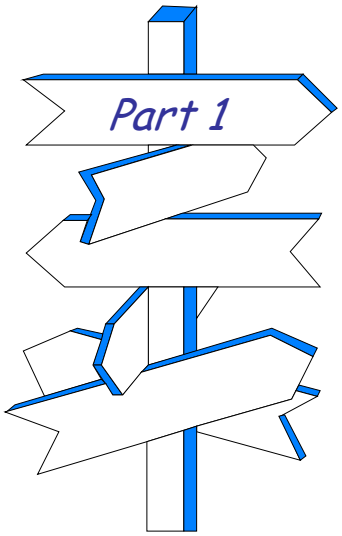
Language requirements for real-time programming

The Java platform

- Overview
- Thread model
- Assessment for real-time applications

Fundamental issues

- Priority inversion and its management
  - Priority Inheritance
  - Priority Ceiling Emulation
- "Garbage Collection"



Requirements for real-time programming (1)

General requirements

- Language features promoting safety/reliability
  - Compile-time checking
  - Encapsulation
  - Deterministic language semantics
- Predictability
  - In deadline-sensitive parts of the program, need to know worst-case execution time for language constructs, API
- Performance (?)

Concurrency

- Adequate range of priority values
- Well-defined semantics for scheduling
- Safe / efficient mutual exclusion, including “state notification”
- Safe / efficient coordination / communication
- Avoidance of unbounded priority inversion
- Support for common real-time idioms such as:
  - Periodic activities
  - Various kinds of events / semaphores

Requirements for real-time programming (2)

Memory management

- Cannot run out of storage or fragment
- Safe/predictable storage reclamation

Asynchrony

- Asynchronous events / event handlers
  - Connection with interrupts
- Asynchronous Transfer of Control
  - Timeout
  - Thread termination

Time

- Support for high-resolution time (millis and nanos), both absolute and relative
- Support for various kinds of timers, clocks

Low-level support

- Access to hardware-specific features

API

- Functionality, “thread safety”

"Java Platform" summary

Java language

- "Pure" Object-Oriented language in the style of Smalltalk
  - Single inheritance of classes, "multiple inheritance" of "interfaces"
- Built-in support for exception handling, threads
- Well-defined semantics, at least for sequential features
- Flexible, dynamic language
  - Classes are run-time objects
  - All non-primitive data go on the heap
- Large emphasis on safety, security (downloadable "applets")
- Garbage collection required

Java Virtual Machine

- Portable, interpretable binary format for Java classes (class file)
- Instruction set tailored to Java language

Java API

- Extensive set of "packages" for a wide variety of application domains
- "Core" libraries, networking, I/O, collections, database, reflection, ...

Overview of Java concurrency support (1)

Unit of concurrency is the *thread*

- An instance of the class **`java.lang.Thread`** or one of its subclasses
- **`run()`** method = algorithm performed by each instance

Either extend Thread, or implement the Runnable interface

- Override/implement `run()`
- All threads are dynamically allocated

Example

```
public class Writer extends Thread{
  final int count;
  public Writer(int count){this.count=count;}

  public void run(){
    for (int i=1; i<=count; i++){
      System.out.println("Hello " + i);
    }
  }
  public static void main( String[] args ) {
    Writer w = new Writer(60);
    w.start();  // New thread of control invokes w.run()
    System.out.println( "In main thread" );  // May be concurrent with w
  } // May return while w is still running
}
```

Overview of Java concurrency support (2)

Lifetime properties

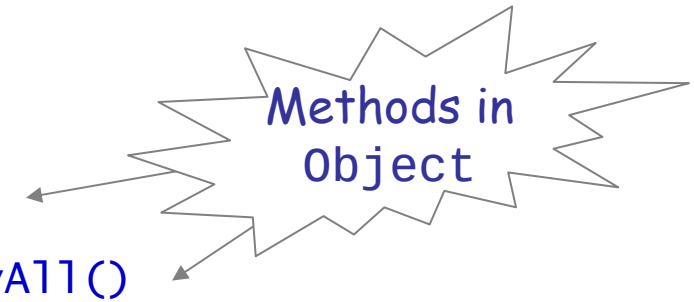
- Constructing a thread creates thread resources (stack, etc.)
- "Activation" of thread `t` is explicit, by invoking `t.start()`
- Started thread runs "concurrently" with parent
- Thread terminates when its `run()` method returns
- Parent does not need to wait for children to terminate

Mutual exclusion

- Shared data (`volatile` fields)
- `synchronized` blocks/methods

Thread coordination / communication

- Pass data to new thread via constructor
- Pulsed event – `obj.wait()` / `obj.notify()`
- Broadcast event – `obj.wait()` / `obj.notifyAll()`
- `t.join()` suspends caller until the target thread `t` completes



Time

- `sleep(millis)`
- Timeout on `obj.wait(millis)`

Overview of Java concurrency support (3)

Scheduling / priorities

- Priority is in range 1..10
- Thread can change/interrogate its own or another thread's priority
- `yield()` gives up the processor

Asynchrony

- `interrupt()` sets a bit that can be polled
- Asynchronous termination
  - `stop()` throws asynchronous exception (deprecated, dangerous)
  - `destroy()` kills a thread (unimplemented, dangerous)
- `suspend()` / `resume()` have been deprecated
- No asynchronous exceptions
- Thread propagating an unhandled exception terminates silently, but first calls its ThreadGroup's `uncaughtException()` method

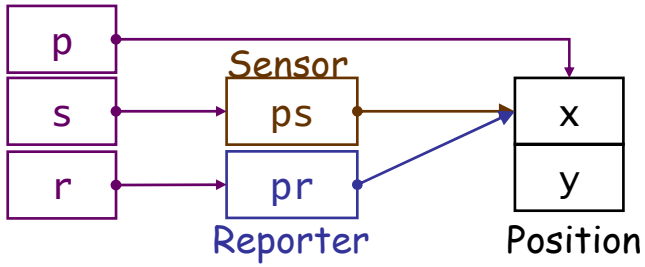
Other functionality

- `ThreadGroup` class
  - Supplies `uncaughtException` method
- Dæmon threads
  - Allows application to terminate though such threads are still running

Mutual exclusion: synchronized blocks

A Position object is periodically updated by a Sensor object and is periodically displayed by a Reporter object

```
class Position{
  double x, y;
}
```

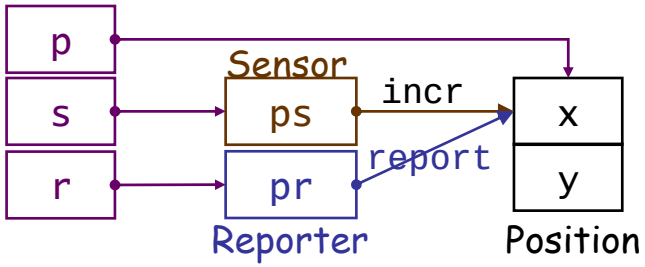



```
class Test{
   public static void main(String[] args){
      Position p = new Position();
      Sensor   s = new Sensor( p );
      Reporter r = new Reporter( p );
      s.start();
      r.start();
   }
}
```

```
class Sensor extends Thread{
  private final Position ps;
  Sensor( Position p ){
    ps=p;
  }
  public void run(){
    while(...){
      synchronized( ps ){
        ps.x += ...;
        ps.y += ...;
      }
      sleep(...);
    }
  }
}
```

```
class Reporter extends Thread{
  private final Position pr;
  Reporter( Position p){
    pr=p;
  }
  public void run(){
    while(...){
      synchronized( pr ){
        System.out.println( pr.x );
        System.out.println( pr.y );
      }
      sleep(...);
    }
  }
}
```

Mutual exclusion: synchronized methods




```
class Position{
  private double x, y;
  public synchronized void incr(double xinc,
                                double yinc){
    x += xinc;
    y += yinc;
  }
  public synchronized void report(){
    System.out.println(x);
    System.out.println(y);
  }
}
```

```
class Sensor extends Thread{
   private final Position ps;
   Sensor( Position p ){
      ps=p;
   }
   public void run(){
      double xinc, yinc;
      while(...){
         xinc = ...;
         yinc = ...
         ps.incr(xinc, yinc);
         sleep(...);
      }
   }
}
```

```
class Reporter extends Thread{
   private final Position pr;
   Reporter( Position p ){
      pr=p;
   }
   public void run(){
      while(...){
         pr.report();
         sleep(...);
      }
   }
}
```

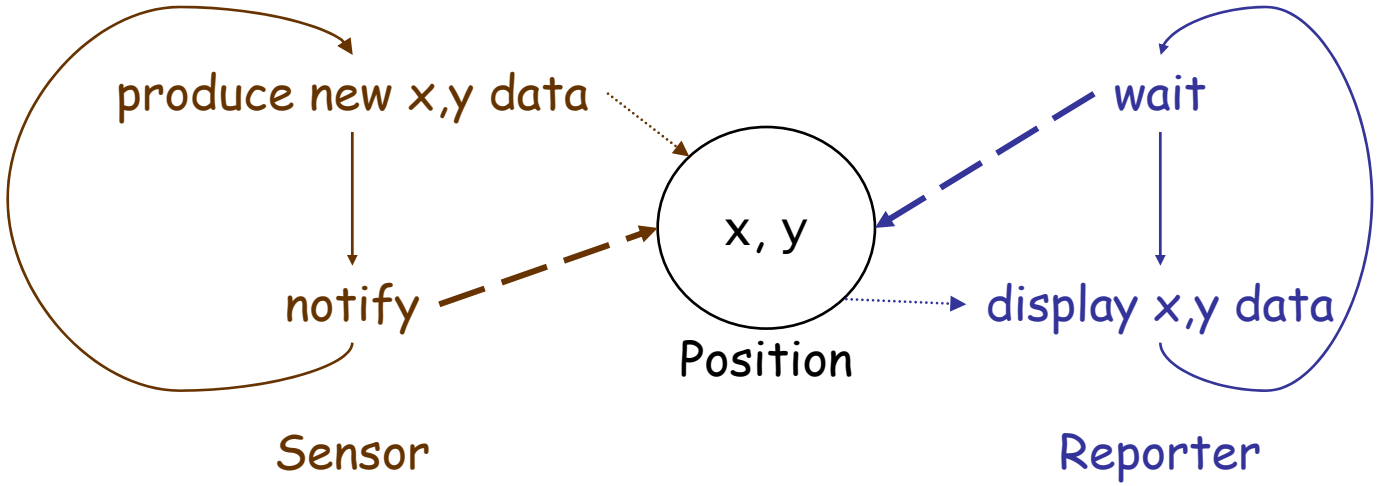
Coordination via wait() and notify() (1)

A Reporter object may display the same x, y pair several times

Variation: "no stale data"

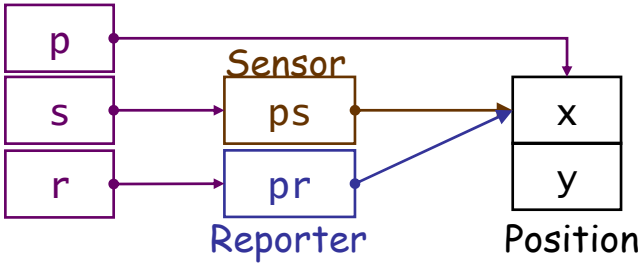
- Ensure that a Reporter always obtains an x, y pair that is set *after* the Reporter requests the data
- Do not care if some x, y pairs go unread
  - Sensor object may generate new values more frequently than the Reporter is retrieving them
  - No need to have a Sensor object wait for the new x, y values to be retrieved, or to buffer them

Schematic



Coordination via wait() and notify() (2)

```
class Position{
  double x, y;
}
```




```
class Test{
   public static void main(String[] args){
      Position p = new Position();
      Sensor   s = new Sensor( p );
      Reporter r = new Reporter( p );
      s.start();
      r.start();
   }
}
```

```
class Sensor extends Thread{
  private final Position ps;
  Sensor( Position p ){
    ps=p;
  }
  public void run(){
    while(...){
      synchronized( ps ){
        ps.x += ...;
        ps.y += ...;
        ps.notify();
      }
      sleep(...);
    }
  }
}
```

```
class Reporter extends Thread{
  private final Position pr;
  Reporter( Position p){
    pr=p;
  }
  public void run(){
    while(...){
      synchronized( pr ){
        pr.wait();
        System.out.println( pr.x );
        System.out.println( pr.y );
      }
      sleep(...);
    }
  }
}
```

AdaCore

Concurrency Utilities in JSR 166

Goals

- Provide high-level, high-performance concurrency building blocks
- Replace many uses of wait / notify

New APIs, based on Doug Lea's `util.concurrent` library

- Mutual exclusion support
  - Explicit locks
  - Explicit condition variables
  - Atomic variables for thread-safe operations with no synchronization
- Communication idioms
  - Multi-way synchronization (rendezvous)
  - Semaphores
- Thread-aware collection classes
- Thread pools
- Thread schedulers
- Nanosecond timing

Part of J2SE 5.0 (Java 1.5)

- Developed independently of Real-Time Specification for Java
- Contained in `java.util.concurrent` package

AdaCore

Why consider Java for real-time?

General benefits

- Language security and (in general) well-defined semantics
- Portability at multiple levels (“Write Once, Run Anywhere”)
- Extensive API

Technical features / expressiveness / flexibility

- Support for software engineering (encapsulation, OOP, exceptions…)
- Built-in feature for concurrency (threads)
- Dynamic loading attractive in some segments such as telecom

Advantages over other languages

- Safer than C
- Simpler than C++
- More popular than Ada

Pragmatics / politics

- Organization adopting Java as an “enterprise” language may wish to use Java for real-time

Critique of Java thread model (1)

Error-prone

- Requires cooperation by the accessing threads
  - Even if all methods are synchronized, an errant thread can access non-private fields without synchronization
- Subtle bug: constructor or synchronized instance method making non-synchronized access to static field
- “Nested monitor” problem

Subtleties in practice

- Not always clear when a method needs to be declared as synchronized
- Complex interactions with other features (e.g. when are locks released)
- Locking is hard to get right (exacerbated by absence of nested objects)

Effect not always clear from source syntax

- A non-synchronized method may be safe to invoke from multiple threads
- A synchronized method might not be safe to invoke from multiple threads

Critique of Java thread model (2)

Thread communication/synchronization issues

- `wait()` and `notify()`/`notifyAll()` are low-level constructs that must be used very carefully
  - "`while (!condition) {obj.wait()}`" needed
- Limited mechanisms for direct inter-thread communication
- Synchronized code that changes object's state must explicitly invoke `notify()` or `notifyAll()`
- No syntactic distinction between signatures of synchronized method that may suspend a caller and one that does not
- Only one wait set per object (versus per associated "condition")

Public thread interface issues

- The need to explicitly initiate a thread by invoking its `start()` method allows several kinds of programming errors
- Although `run()` is part of a thread class's public interface, invoking it explicitly is generally an error

"Inheritance Anomaly"

- Subclass may need to redo superclass synchronization logic

Problems with Java for real-time (1)

Lack of some features useful for software engineering

- Operator overloading, strongly typed primitive types, …

Thread model deficiencies

- Priority range (1..10) too narrow
- Priority semantics are implementation dependent and fail to prevent unbounded priority inversion
- Relative `sleep()` not sufficient for periodicity

Memory management unpredictability

- Predictable, efficient garbage collection appropriate for real-time applications is not (yet) in the mainstream
- Java lacks stack-based objects (arrays and class instances)
- Heap used for exceptions thrown implicitly as an effect of other operations

Asynchrony deficiencies

- Event handling requires dedicated thread
- `interrupt()` not sufficient
- `stop()` and `destroy()` deprecated or dangerous or both

Regular Java semantics for scheduling

Section 17.12 of the Java Language Specification

- "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. *Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.*"

Problems for real-time applications

- Impossible to guarantee that deadlines will be met for periodic threads
  - May get priority inversion
- No guarantee that priority is used for selecting a thread to unblock when a lock is released
- No guarantee that priority is used for selecting which thread is awakened by a `notify()`, or which thread awakened by `notifyAll()` is selected to run

Problems with Java for real-time (2)

Run-time issues

- Dynamic class loading is expensive, not easy to see when it will occur
- Array initializers ⇨ run-time code
  - `int[] arr = {10, 20, 30};`

OOP has not been embraced by the real-time community

- Dynamic binding complicates analyzability
- Garbage Collection defeats predictability

Lack of features for accessing the underlying hardware

Performance questions

API Issues

- "Standard" API would need to be rewritten for predictability
- A class's "interface" is more than its public and protected members
  - In general it includes some implementation characteristics
  - E.g. does it allocate objects, can it block

Some JVM opcodes require non-constant amount of time

AdaCore

Periodic threads in Java?

```
class Periodic extends Thread{
  static interface Action{ void doIt(); }
  private final Action action;
  private final long periodMillis;
  Periodic(String name, long periodMillis, int priority, Action action){
    super(name);
    this.periodMillis = periodMillis;
    this.action = action;
    this.setPriority(priority);
  }
  public void run(){
    long nextTime = System.currentTimeMillis();
    while(true){
      this.action.doIt(); // action to be performed
      nextTime += periodMillis;
      try{
        sleep(nextTime - System.currentTimeMillis());
      } catch (InterruptedException e){ return; }
    }
  }
  public static void main(String[] args){
     Periodic p1 = new Periodic("High", 100, 10, new Action(){...});
     Periodic p2 = new Periodic("Low", 500, 2, new Action(){...});
     p1.start(); p2.start(); ...
     p1.interrupt(); p2.interrupt();
  }
}
```

Problems:

- No guarantee that priorities will be obeyed
- Relative delay in sleep may lead to missed deadlines

Periodicity via Timer and TimerTask?

java.util.TimerTask

- A Runnable object whose run() method is invoked, either once or repeatedly, by a Timer

java.util.Timer

- An object with an associated background thread that invokes the Timer's TimerTasks

```
import java.util.*;
class PeriodicExample{
  public static void main(String[] args){
    Timer periodic1 = new Timer();
    Timer periodic2 = new Timer();
    periodic1.scheduleAtFixedRate(
      new TimerTask(){
        public void run(){...} // periodic action 1
      }, 0, 100); // start now, period = 100ms
    periodic2.scheduleAtFixedRate(
      new TimerTask(){
        public void run(){...} // periodic action 2
      }, 0, 500); // start now, period = 500ms
    ...
    periodic1.cancel(); // cancels when run() returns
    periodic2.cancel(); // cancels when run() returns
  }
}
```

Problems:

- No way to associate priorities with Timers
- SDK 1.3 documentation warns that Timer "does not offer real-time guarantees: it schedules tasks using the Object.wait(long) method"

Priority inversion - introduction

What is a "priority inversion"?

- A situation in which a higher-priority thread is blocked/stalled involuntarily while a lower-priority thread is running

It is sometimes necessary (to enforce mutual exclusion)

- When the lower priority thread holds a lock that is needed by the higher priority thread

Scheduling policy affects the worst case blocking time

- Under some policies, a high priority thread may be blocked (stalled on a lock) during the execution of a lower-priority thread that does not hold the lock - *unbounded priority inversion*
- Priority Inheritance and Highest Lockers (Priority Ceiling) reduce this time considerably

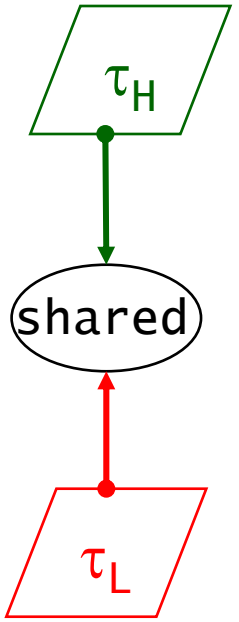
Priority inversion example (1)

```
class High extends Thread{
  private Object ref;
  High(Object ref){
    this.setPriority(10);
    this.ref=ref;
  }
  public void run(){  // periodic
    ... synchronized(ref){...} ...
  }
}

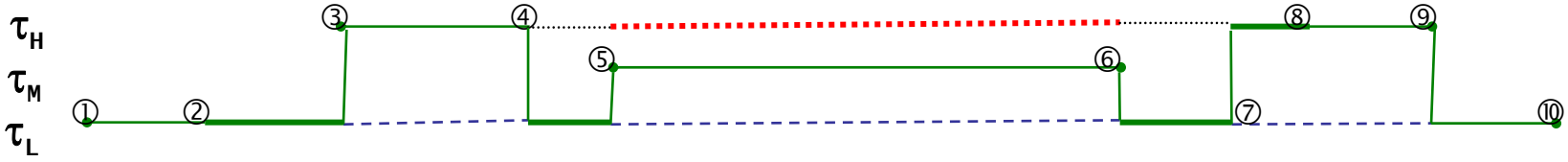
class Low extends Thread{
  private Object ref;
  Low(Object ref){
    this.setPriority(1);
    this.ref=ref;
  }
  public void run(){  // periodic
    ... synchronized(ref){...} ...
  }
}

class Medium extends Thread{
  Medium(){ this.setPriority(5); }
  public void run(){ ... }
}
```

```
class PriorityInversion{
  public static void main(String[] args){
    Object shared = new Object();
    High τH = new High(shared);
    τH.start();
    Low  τL = new Low(shared);
    τL.start();
    Medium τM = new Medium();
    τM.start();
  }
}
```



Priority inversion example (2)



τ_H has high priority, τ_M medium priority, and τ_L low priority

① τ_L awakens and starts to run (the other two threads are blocked, waiting for the expiration of delays)

② τ_L starts to use a mutually-exclusive resource

- Enters a monitor, locks a semaphore

③ τ_H awakens and preempts τ_L

④ τ_H tries to use the resource held by τ_L and is blocked, so τ_L resumes

- This priority inversion is necessary

⑤ τ_M awakens and preempts τ_L

- "Unbounded" priority inversion: τ_M indirectly prevents τ_H from running

⑥ τ_M completes, and τ_L resumes

⑦ τ_L releases the resource, is preempted by τ_H, which uses the resource

⑧ τ_H releases the resource

⑨ τ_H completes execution, allowing τ_L to resume

⑩ τ_L completes execution

Priority inversion reduction mechanisms

Priority Inheritance

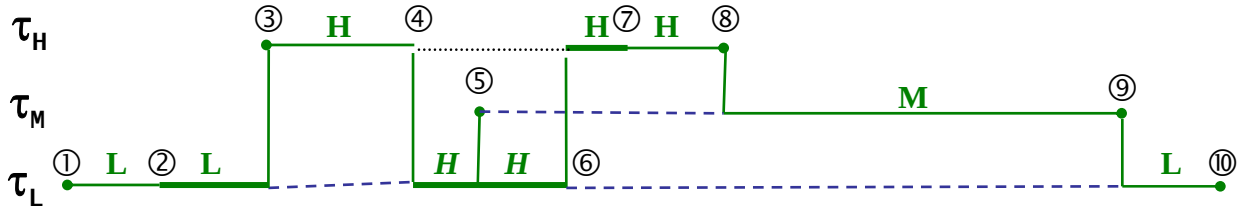
- When a thread τ_H attempts to acquire a lock that is held by a lower-priority thread τ_L, τ_L *inherits* τ_H's priority as long as it is holding the lock
  - Thus a medium priority thread τ_M cannot preempt τ_L while τ_L is holding the lock required by τ_H
- Applied transitively if τ_L is waiting for a lock held by a yet-lower-priority thread

Highest Lockers (Priority Ceiling Emulation)

- Each lockable object has a *ceiling priority* equal to the highest priority of any thread that could lock the object
- When a thread acquires a lock, its priority is boosted to the object's ceiling
  - Run-time error if the thread's priority exceeds the object's ceiling
- When it releases the lock, its priority is reset
- Thus while holding a lock, a thread executes at a priority higher than or equal to that of any thread that needs the lock

AdaCore

Priority Inheritance example



① τ_L awakens and starts to run at priority L

② τ_L starts to use a mutually-exclusive resource

③ τ_H awakens, preempts τ_L and runs at priority H

④ τ_H tries to use the resource held by τ_L and is blocked, τ_L resumes

- At this point τ_L inherits τ_H's priority (H)

⑤ τ_M awakens *but does not preempt* τ_L

- This avoids the unbounded priority inversion

⑥ τ_L releases the resource, reverts to its pre-inheritance priority L, and is preempted by τ_H, which can then use the resource

⑦ τ_H releases the resource

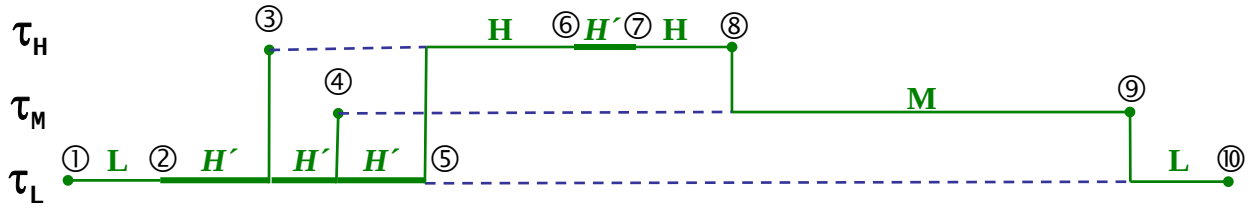
⑧ τ_H completes execution, allowing τ_M to execute

⑨ τ_M completes, allowing τ_L to resume

⑩ τ_L completes execution

Effect: a thread holding a lock executes at the max priority of all threads currently requiring that lock

Priority Ceilings ("Highest Lockers") example



① τ_L awakens and starts to run at priority L

② τ_L starts to use a mutually-exclusive resource with ceiling H' ≥ H, and runs at priority H'

- This will prevent unbounded priority inversion

③ τ_H awakens *but does not preempt* τ_L

④ τ_M awakens *but does not preempt* τ_L

⑤ τ_L releases the resource, reverts to its pre-ceiling priority L, and is preempted by τ_H which then runs at priority H

⑥ τ_H starts using the resource with ceiling H' ≥ H, runs at priority H'

⑦ τ_H releases the resource and reverts to priority H

⑧ τ_H completes execution, allowing τ_M to execute

⑨ τ_M completes, allowing τ_L to resume

⑩ τ_L completes execution

Effect: a thread holding a lock executes at a priority no less than that of any thread that might need the lock

AdaCore

Priority Inheritance

Advantages

☺ Supported by many RTOSes

☺ Only change priority when needed (thus no cost in common case when resource not in use)

Disadvantages

☹ Task may be blocked once for each shared resource that it needs ("chained blocking")

☹ Implementation may be expensive

- Task's priority is changed by an action external to the task

Other properties

- Sacrifices responsiveness for predictability
  - A task may be prevented from running in order to guarantee that deadlines are met overall

Priority Ceiling Emulation

Advantages

☺ If no thread can block while holding the lock on a given shared object, then a queue is not needed for that object

- ☺ In effect, the processor is the lock
- ☺ Prevents deadlock (on uniprocessor)

☺ Ensures that a task is blocked only once each period, by one lower priority task holding the lock

Disadvantages

☹ Fixed ceilings not appropriate for applications where priorities need to change dynamically

☹ Requires check and priority change at each call

- Overhead even if object not locked
- But this is inconsequential in the queueless case

Other properties

- If ceiling high, effect ≅ disabling task switching
- Sacrifices responsiveness for predictability

Garbage collection

There is a potential issue of storage leakage on assignment to a variable of a reference type

- May make inaccessible the object previously referenced

```
while(...){ Point p = new Point(0, 0);... }
```

In many languages this would be a problem

- Solutions involve unchecked storage reclamation, or provision of a finalization operation for the type/class

The JVM performs automatic reclamation ("garbage collection")

- There is no language facility to explicitly deallocate an object
- The garbage collector operates (conceptually) as a background thread
  - It generally executes only when the program is otherwise suspended (e.g. waiting for user input)
  - It may also be invoked implicitly if there is not enough storage to fulfill an allocation request
- If you want to indicate that a reference to an object is no longer needed, then assign `null` to the variable
- You can also invoke the garbage collector directly

Garbage collection technology

Main concepts

- At any point, the values that can be manipulated are the *roots* (stack, static variables) and the *live* heap objects
  - *Live* = directly or indirectly accessible from a root
- *Free space* = heap locations available for future allocations
- *Garbage* = objects that are neither live nor free

General issues related to dynamic memory management

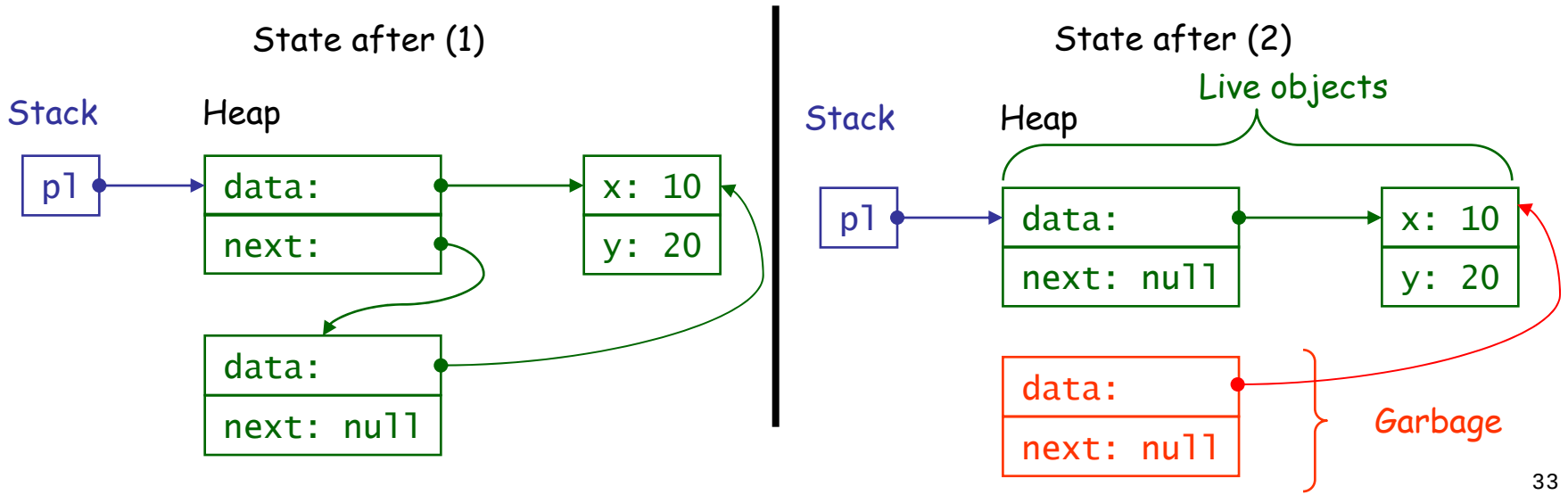
- Storage leakage if garbage is generated and not freed
- Fragmentation if free space is not compacted
- Dangling reference if a live object is freed

Potential costs associated with garbage collection

- Time
  - Reclaiming objects, allocating objects
  - Assigning to a pointer, reading the value of a pointer
  - Delays imposed by the execution of the collector
- Space
  - Per heap object
  - Overhead intrinsic to collector algorithm

Example of live objects and garbage

```
class PointList{
  Point data;
  PointList next;
  PointList(Point data, PointList next){
     this.data=data;
     this.next=next;
  }
  public static void main(String[] args){
    PointList pl =
      new PointList(new Point(10, 20), null);
    pl.next =
      new PointList( pl.data, null); // (1)
    pl.next = null;                  // (2)
  }
}
```



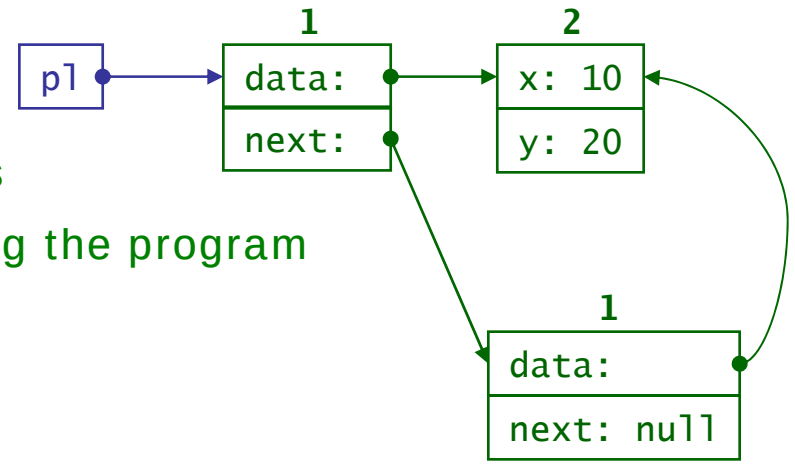
Reference counting

Basics

- Each object maintains a count of the number of pointers to that object, from roots or heap objects
- When a reference count is 0, the object can be freed

Advantages

- Conceptually simple
- Immediate reuse of short-lived objects
- Distributes overhead versus suspending the program



Disadvantages

- Does not detect cyclic structures
- Space penalty on each object
- Overhead on object allocation and pointer assignment to maintain the reference count
- Need special processing to avoid unbounded trace of an object whose count goes to 0
- Fragmentation

"Mark-sweep" collector

Basics

- *Mark* phase: global traversal of all live objects, starting from roots
- *Sweep* phase: any heap locations not marked are freed

Advantages

- Handles cycles
- No overhead on pointer manipulation
- No extra space per object

Disadvantages

- Computation halts during collection ⇨ not for real-time
- Fragmentation unless combine with compaction
- Sweep phase time proportional to size of heap
- Thrashing if heap occupancy high

"Copying" collectors

Basics

- Heap comprises two semi-spaces, one containing current data and the other containing obsolete data
- Collector flips the two spaces, traverses the active objects in old space (*Fromspace*) and copies them into new space (*Tospace*), preserving object topology (sharing)

Advantages

- Free space is linear
  - Allocation cost low
  - Variable-sized objects do not present a problem
- Automatically compacts (avoids fragmentation)
- Cost proportional to live data versus whole heap (but see below)

Disadvantages

- Need to reserve twice the space actually needed
- Copying is more expensive than bit marking in Mark-Sweep
- Long-lived data objects are copied repeatedly
- Computation halts during collection ⇨ not for real-time

AdaCore

Generational collectors

Basics

- "Weak generational hypothesis": most objects die young
  - Focus reclamation on the objects created most recently
  - Collection triggered when number of bytes allocated since previous collection exceeds some threshold
- Segregate objects by age into heap regions (generations)
- Younger generation collected more frequently than older
- Young objects that survive some number of collections (typically 1 or 2) can be promoted to an older generation
- Most generational algorithms use copying, but mark-sweep possible

Advantages

- Pauses shorter since less data traced and copied
- Collection may be scheduled to minimize program disruption

Disadvantages

- Determining roots is more difficult, since pointers (especially from old to young) may be inter-generational
  - "Write barrier" overhead on pointer assignment
- "Tenured garbage" not reclaimed immediately

Incremental and concurrent collection

Goal is to guarantee *worst-case* pause time, for usability in real-time applications

- Generational technique tries to optimize *average* pause time

***Incremental* collector is interleaved with user program**

- Example: reference counting

***Concurrent* collector runs as a separate thread**

In both cases, collector must ensure heap state consistency

- The collector may temporarily treat some garbage as live ("floating garbage")
- "Write barriers" typical with mark-sweep collectors
- "Read barriers" typical with copying collectors

User needs to ensure that garbage is not generated more quickly than it can be collected

In principle, an incremental or concurrent collector can be used in a real-time program

- But hardware support has been required for adequate performance

Garbage collection and real-time programming

No Garbage Collection

- Require that all allocations be performed at system initialization
- Common in many kinds of real-time applications
- Difficult in Java: all non-primitive data are dynamically allocated

Real-Time Garbage Collector

- Techniques exist that have predictable / bounded costs
- But programmer still needs to ensure that allocation rate does not exceed rate at which GC can reclaim space
- Also, in the absence of specialized hardware, such techniques tend to introduce high latencies
  - GC needs to run at high priority, since if it is preempted while the heap is in an inconsistent state, bad things can happen

Hybrid approach

- For low latency, allow a thread to preempt GC if the thread never references the heap
  - In general, need run-time check on each heap reference
- Allow a thread to allocate objects in a scope-associated area
  - Area flushed at end of scope/thread

Part 2: Real-Time Specification for Java

Overview and design goals

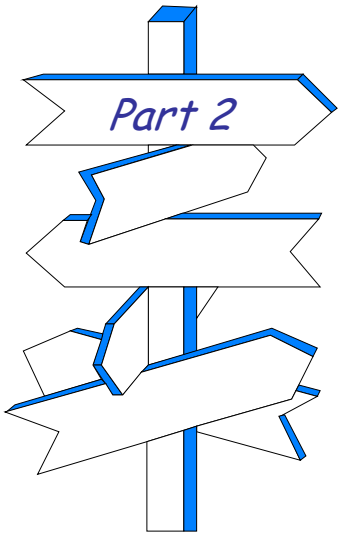
- Java Community Process
- History/status of real-time Java efforts
- Guiding principles
- Summary

Scheduling

- Class summary
- RealtimeThread properties
- Base scheduler
- Critique

Memory Management

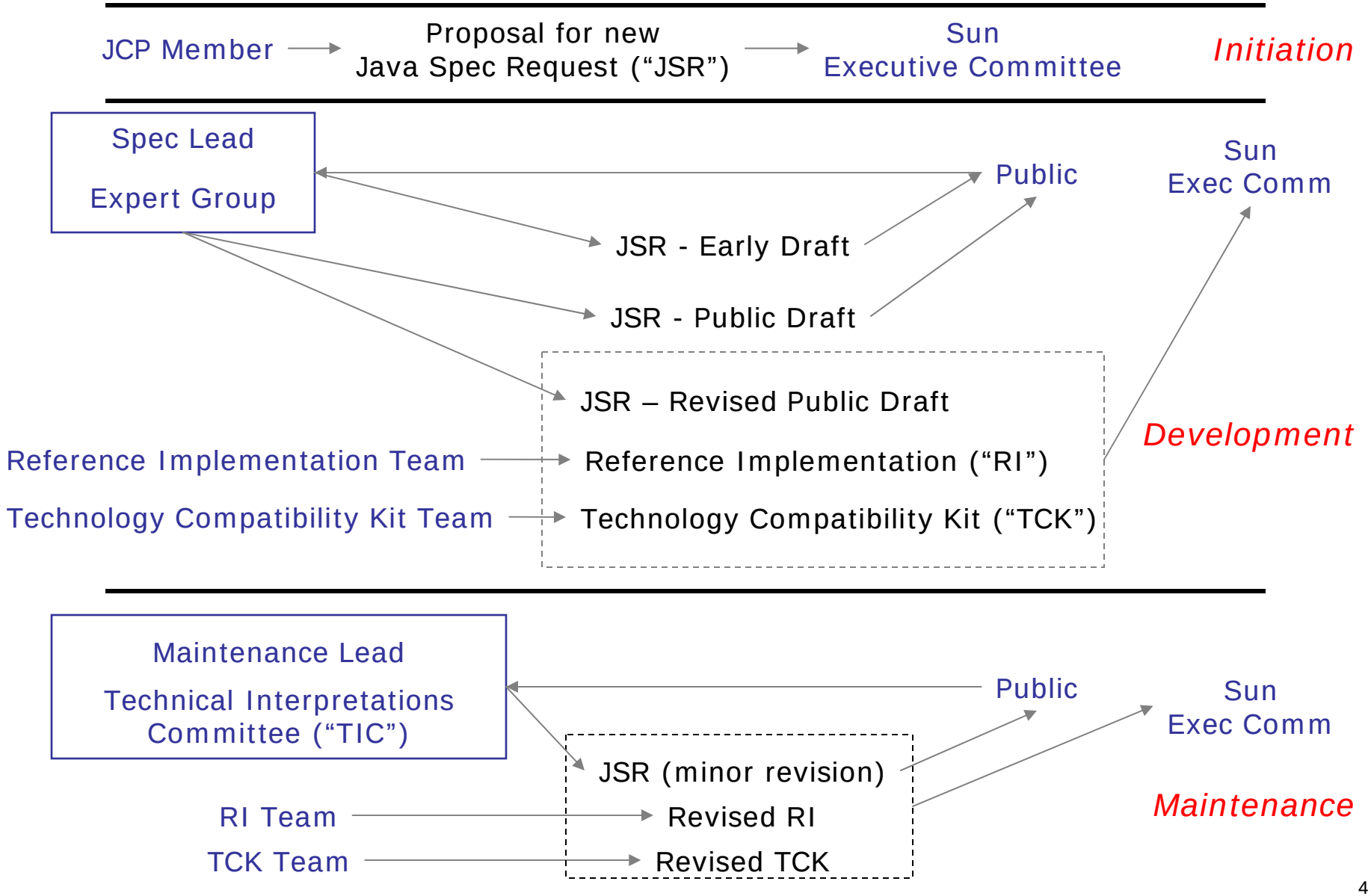
- Memory areas – general properties
- Programming with “scoped memory”
- Critique



Java Community Process ("JCP")

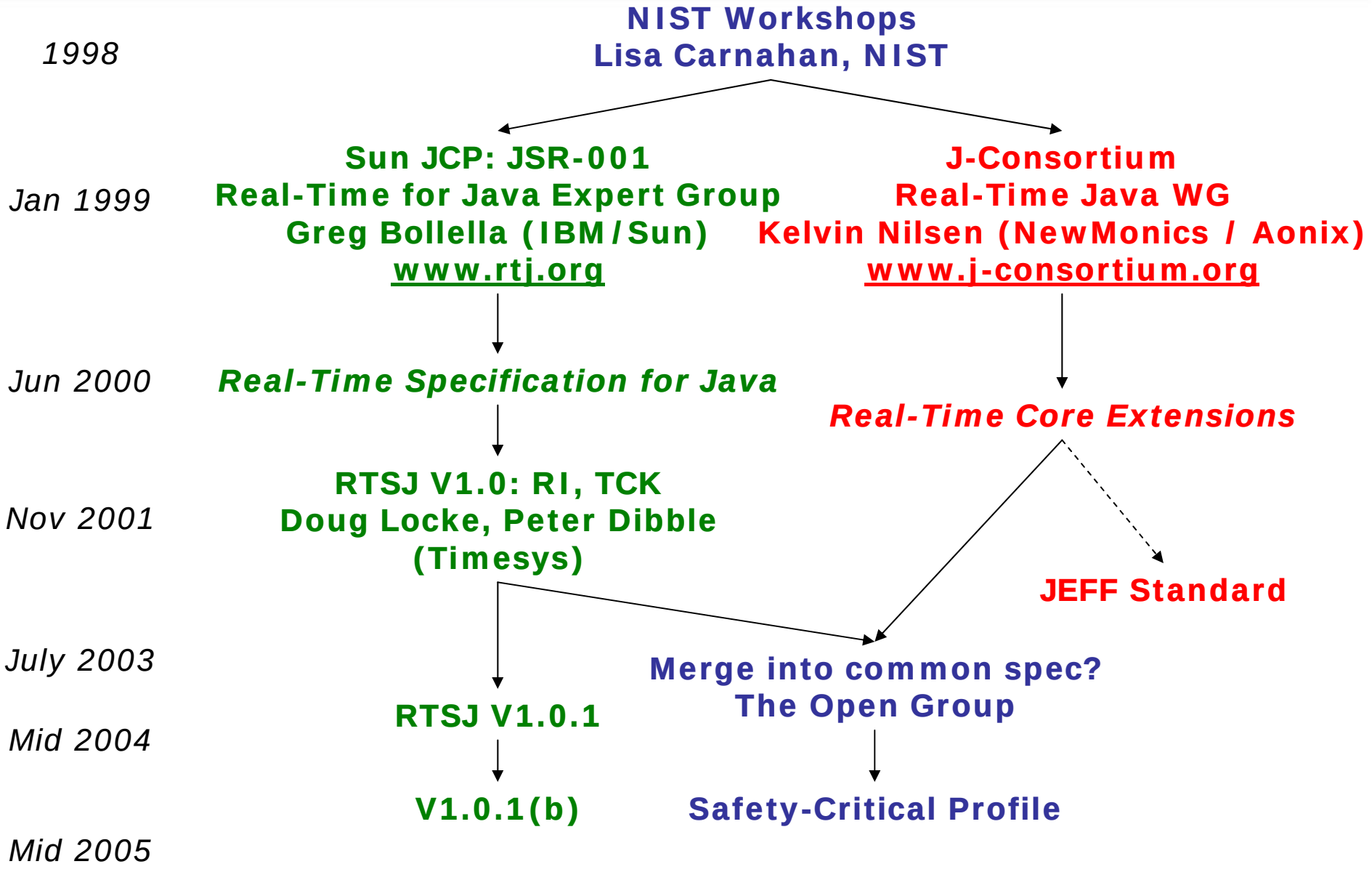
Sun-administered process for augmenting / modifying the Java platform

www.jcp.org/en/procedures/jcp2



AdaCore

History of real-time Java efforts



Focus here will be on the Real-Time Specification for Java

AdaCore

Contributors to the RTSJ

Initial development (February 1999 – June 2000)

"Real-Time for Java Expert Group", JSR-001

Greg Bollella (IBM / Sun)
James Gosling (Sun)
Ben Brosgol (Aonix / AdaCore)
David Hardin (Rockwell Collins / aJile)
Peter Dibble (Microware / Timesys)
Mark Turnbull (Nortel Networks, Canada)
Steve Furr (QSSL, Canada)

Completion of V1.0 (July 2000 – November 2001)

Peter Haggar (IBM)
Doug Locke (Timesys)
Peter Dibble (Timesys)
Ben Brosgol (AdaCore)
Greg Bollella (Sun)
Rudy Belliardi (Schneider Automation)
Timesys Reference Implementation Team

Spec maintenance, RTSJ Technical Interpretation Committee (Nov. 2001- present)

Peter Dibble (Timesys)
Rudy Belliardi (Schneider Automation)
Greg Bollella (Sun)
Andy Wellings (Univ. of York, UK)
Ben Brosgol (AdaCore)
David Holmes (DLTeCH, Australia)

Guiding principles for the RTSJ

Applicability to particular Java environments

- Usage not restricted to particular versions of the JDK

Backward compatibility

- Existing Java code can run on RTSJ implementations

"Write Once, Run Anywhere"

- Important goal but difficult to achieve for real-time programs

Current practice vs. advanced features

- Address current real-time practice but allow implementations to include advanced features

Execution predictability

- Highest priority goal, may require compromise of performance/throughput

No syntactic extension

- No new keywords or other non-standard syntax

Allow variation in implementation decisions

- Recognize that implementations differ in terms of tradeoffs
- Do not mandate specific algorithms

Summary of main RTSJ features

Concurrency

- Class `RealtimeThread` extends `java.lang.Thread`
- Flexible scheduling framework, base scheduler (fixed-priority preemptive)

Memory management

- "Immortal", "Scoped" memory areas augment garbage-collected heap
- "NoHeap realtime thread" can preempt GC

Synchronization (priority inversion management)

- Priority inheritance (required), priority ceiling emulation (optional)

Asynchrony

- Asynchronous Event Handling
- Asynchronous Transfer of Control

Time and timers

- High-resolution time (absolute, relative)
- Timers (periodic, one-shot)

Low-level features

- Specialized kinds of "physical" memory
- "Peek/poke" of primitive data in "raw" memory

Concurrency support in the RTSJ

General concept of *schedulable object*

- Realtime thread or asynchronous event handler
- Arguments to constructor establish various "parameters"
  - *Scheduling parameters* (e.g., priority)
  - *Release parameters* (cost, deadline, overrun/miss handlers)
    - Periodic parameters (start, period)
    - Aperiodic parameters
      - Sporadic parameters (minimum interarrival time)
  - *Memory parameters* (e.g., rate of allocation on heap)

Initial default scheduler ("base scheduler")

- Must support at least 28 distinct priority values, beyond Java's 10
- Preemptive, fixed priority, FIFO within priority

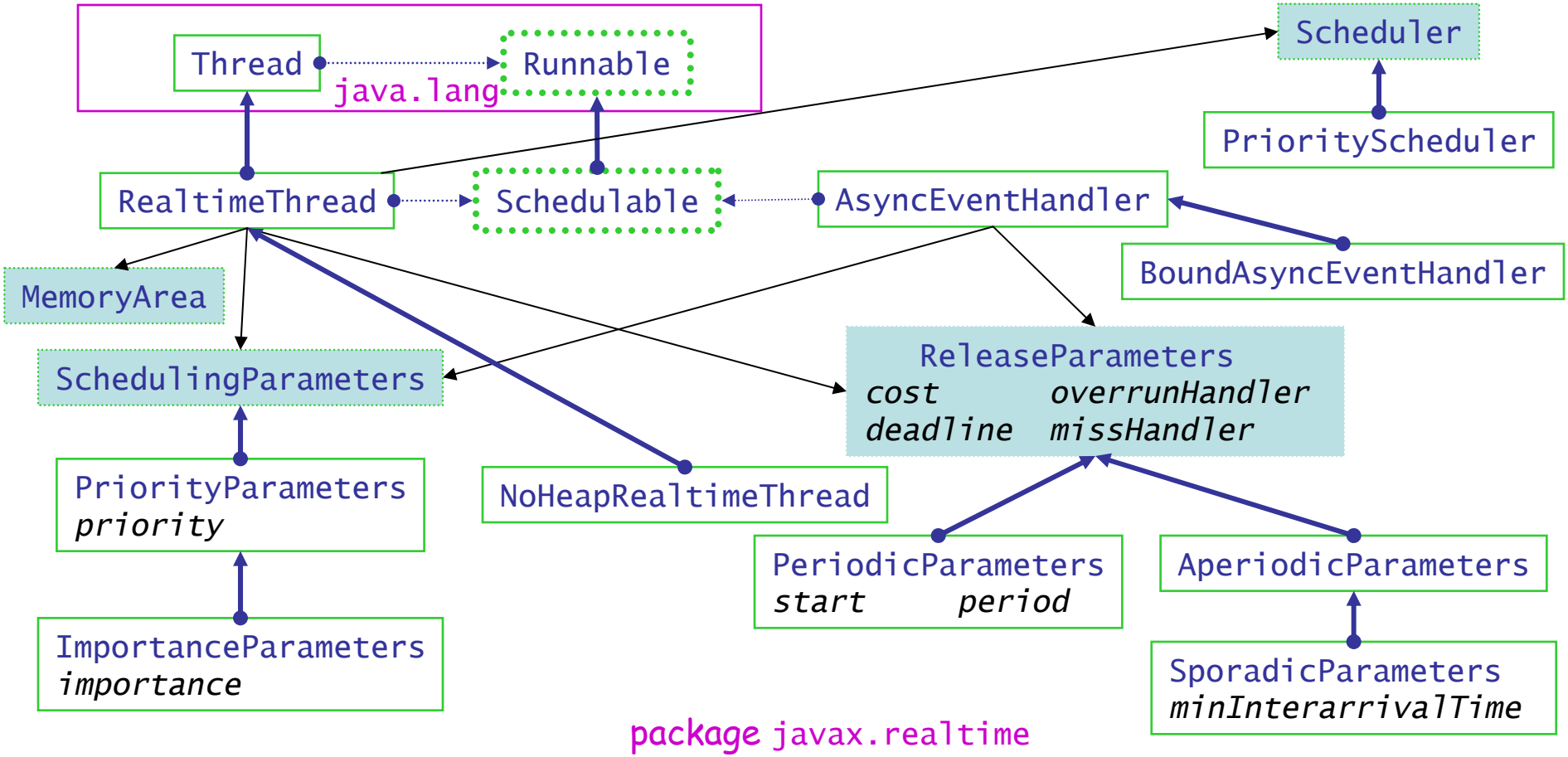
Support for on-line feasibility analysis (optional)

- Implementation can query release parameters to determine if a set of schedulable objects can satisfy some constraint

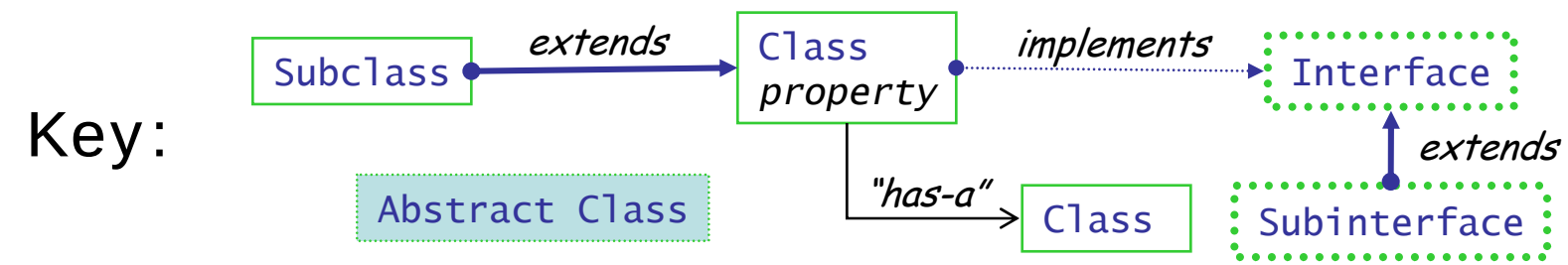
Flexibility

- Implementation can install arbitrary scheduling algorithms
- Users can replace these dynamically, can have different schedulers for different schedulable objects

Scheduling-related classes (partial list)



Key:



Construction of parameters objects

Scheduling parameters

- `new PriorityParameters( 30 );`
- `new PriorityParameters(`
  `PriorityScheduler.instance().getMaxPriority() )`

Release parameters

- ```
  new PeriodicParameters(
      null,                          // start time => now
      new RelativeTime( 100, 0 ),    // period => 100 ms, 0 ns
      new RelativeTime(  20, 0 ),    // cost => 20 ms
      new RelativeTime(  80, 0 ),    // deadline => 80 ms
      null,                          // no cost overrun handler
      null )                         // no deadline miss handler
  ```
- `new SporadicParameters( minInterarrival, cost, deadline,`
  `overrunHandler, deadlineMissHandler)`

Memory parameters

- ```
  new MemoryParameters(
       100000,  // max bytes allocatable in initial memory area
        50000,  // max bytes allocatable in immortal memory
         1000 ) // max bytes per second on the heap
  ```

AdaCore

Example: creating a periodic real-time thread

```
public class MyPeriodic extends RealtimeThread{
  public MyPeriodic( int priority,
                     int periodMillis,
                     int costMillis,
                     int deadlineMillis ){
    super( new PriorityParameters( priority ), // SchedulingParameters
           new PeriodicParameters(             // ReleaseParameters
             null, // 1st release is at start
             new RelativeTime( periodMillis, 0 ),
             new RelativeTime( costMillis, 0 ),
             new RelativeTime( deadlineMillis, 0 ),
             null, // No cost overrun handler )
             null) // No deadline miss handler
    );
  }
  public void run(){
    while (true) {
      … // Work done during each release
      RealtimeThread.waitForNextPeriod();  // Block till next release
    }
}

MyPeriodic mp = new MyPeriodic( 30, 100, 20, 100 );
mp.start();  // Triggers initial release for mp
```

Methods in the Schedulable interface

Methods related to feasibility analysis

```
public boolean addToFeasibility()
public boolean removeFromFeasibility()
```

- Include / exclude this schedulable object from the Scheduler's feasibility analysis

Methods for accessing "parameters" objects

- "Getter/setter" methods for SchedulingParameters, ReleaseParameters, MemoryParameters

Methods for accessing a scheduler

```
public Scheduler getScheduler()
public void setScheduler(Scheduler scheduler)
```

- Get / set the Scheduler for this schedulable object

The run() method

- The executable logic for this schedulable object

The RealtimeThread class (1)

Extends java.lang.Thread and implements Schedulable

Constructors

- General-purpose constructor

```
public RealtimeThread( SchedulingParameters        scheduling,
                       ReleaseParameters           release,
                       MemoryParameters            memory,
                       MemoryArea                  area,
                       ProcessingGroupParameters   group,
                       java.lang.Runnable          logic)
```

- Constructors with default null values, interpreted based on default Scheduler at the time of the RealtimeThread's construction

```
public RealtimeThread()

public RealtimeThread(SchedulingParameters scheduling)

public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters    release)
```

The RealtimeThread class (2)

Methods relevant to periodic RealtimeThreads

- Suspend until next period

```
public static boolean waitForNextPeriod()
public static boolean waitForNextPeriodInterruptible()
```

  - If not in deadline miss condition
    - Block until next release (start of next period)
    - Then return `true`
  - If in deadline miss condition (base scheduler semantics, simplified)
    - If a deadline miss handler has been provided, it is released when the miss occurs
    - Whether or not deadline miss handler has been provided, block until released
      - If no handler: start of next period
      - If handler: later of when handler calls `schedulePeriodic()` or start of next period
    - Then return `false`
  - Interruptible version allows exceptional awakening from blocked state
- Begin / stop unblocking the target periodic RealtimeThread

```
public void schedulePeriodic()
public void deschedulePeriodic()
```

  - Generally called from asynch event handler for deadline miss or cost overrun

The RealtimeThread class (3)

Relative and absolute sleep() method

```
public static void sleep( HighResolutionTime time )
```

- HighResolutionTime has subclasses AbsoluteTime and RelativeTime
- Another sleep method takes a Clock parameter
- Not needed for periodic RealtimeThreads

Other methods

- RealtimeThread analog to Thread.currentThread()

```
public static RealtimeThread currentRealtimeThread()
```

- RealtimeThread overriding of Thread.interrupt()

```
public void interrupt()
```

  - To be discussed below (Asynchronous Transfer of Control)

Cost enforcement (optional)

- If schedulable object t exceeds cost, the following occurs (somewhat simplified)
  - Handler, if any, is released (it may increase budget)
  - Base scheduler semantics: t is suspended until its next release event, at which time it is given a fresh budget and continues

ReleaseParameters subclasses (1)

ReleaseParameters is abstract superclass of several classes that reflect different kinds of release characteristics

- A ReleaseParameters instance is bound to a schedulable object at construction time, may be replaced or updated in place dynamically

PeriodicParameters

- Constructor

```
public PeriodicParameters( HighResolutionTime      start
                           RelativeTime            period
                           RelativeTime            cost
                           RelativeTime            deadline
                           AsynchEventHandler      overrunHandler
                           AsynchEventHandler      missHandler )
```

Released if cost overrun

Released if deadline miss

- Methods
  - "Getter" and "setter" methods for attributes
  -
```
public boolean setIfFeasible(
  RelativeTime period, RelativeTime cost, RelativeTime deadline)
```

- Usage
  - Realtime thread constructed with PeriodicParameters has run() method that loops on waitForNextPeriod or waitForNextPeriodInterruptible
  - Release is implicit, managed by implementation

ReleaseParameters subclasses (2)

AperiodicParameters

- Especially useful for AsyncEventHandlers whose releases are not periodic
- Need a queue (of event fire times) for pending releases
- Constructor
  - ```
    public AperiodicParameters( RelativeTime        cost
                                RelativeTime        deadline
                                AsynchEventHandler  overrunHandler
                                AsynchEventHandler  missHandler )
    ```
- Methods
  - ```
    public void setInitialArrivalTimeQueueLength( int initial )
    ```
  - Getters and setters for `ArrivalTimeQueueOverflowBehavior`
    - Throw exception, ignore, replace, or save
  - ```
    public boolean setIfFeasible( RelativeTime cost,
                                  RelativeTime deadline )
    ```
- Usage
  - AsynchEventHandler constructed with an `AperiodicParameters` has `handleEvent()` method that responds to the event
  - Release occurs when the event is fired
- In a future version of the RTSJ, `RealtimeThread` may have a `waitForNextRelease` method, relevant for `AperiodicParameters`
    ```

ReleaseParameters subclasses (3)

SporadicParameters (subclass of AperiodicParameters)

- Associate with aperiodic schedulable objects with a minimum interarrival time between releases
- Constructor
  - ```
    public SporadicParameters( RelativeTime       minInterarrival,
                               RelativeTime       cost
                               RelativeTime       deadline
                               AsynchEventHandler overrunHandler
                               AsynchEventHandler missHandler )
    ```
- Methods
  - ```
    public void setMinimumInterarrival( RelativeTime minimum )
    ```
  - Getters and setters for `MitViolationBehavior`
    - Throw exception, ignore, replace, or save
  - ```
    public boolean setIfFeasible( RelativeTime cost,
                                  RelativeTime deadline )
    ```
- Usage
  - Same style as for `AperiodicParameters`
    ```

The MemoryParameters class

Purpose

- Admission control for feasibility analysis
- Control over garbage collector pacing
- Control over storage usage by associated schedulable objects

Constructors

- ```
  public MemoryParameters(
      long maxMemoryArea,   // limit on # bytes in initial memory area
      long MaxImmortal,     // limit on # bytes in immortal memory
      long allocationRate ) // limit on # bytes / second
  ```

- ```
  public MemoryParameters( long maxMemoryArea,
                           long MaxImmortal )
  ```

Methods

- Getters and setters for maxMemoryArea, maxImmortal, allocationRate
- `public boolean setMaxMemoryAreaIfFeasible( long maximum )`
- `public boolean setMaxImmortalIfFeasible( long maximum )`
- `public boolean setAllocationRateIfFeasible(long allocationRate)`

The ProcessingGroupParameters class

Purpose

- Assign maximum execution time budget per “period”, collectively to a group of schedulable objects (but cost enforcement is an optional feature)
- Allows inclusion of group of aperiodic schedulable objects in feasibility analysis, but may also apply to periodic and sporadic
- Budget replenished at the start of each “period”
- Generalization of classical “deferrable server”

Constructor

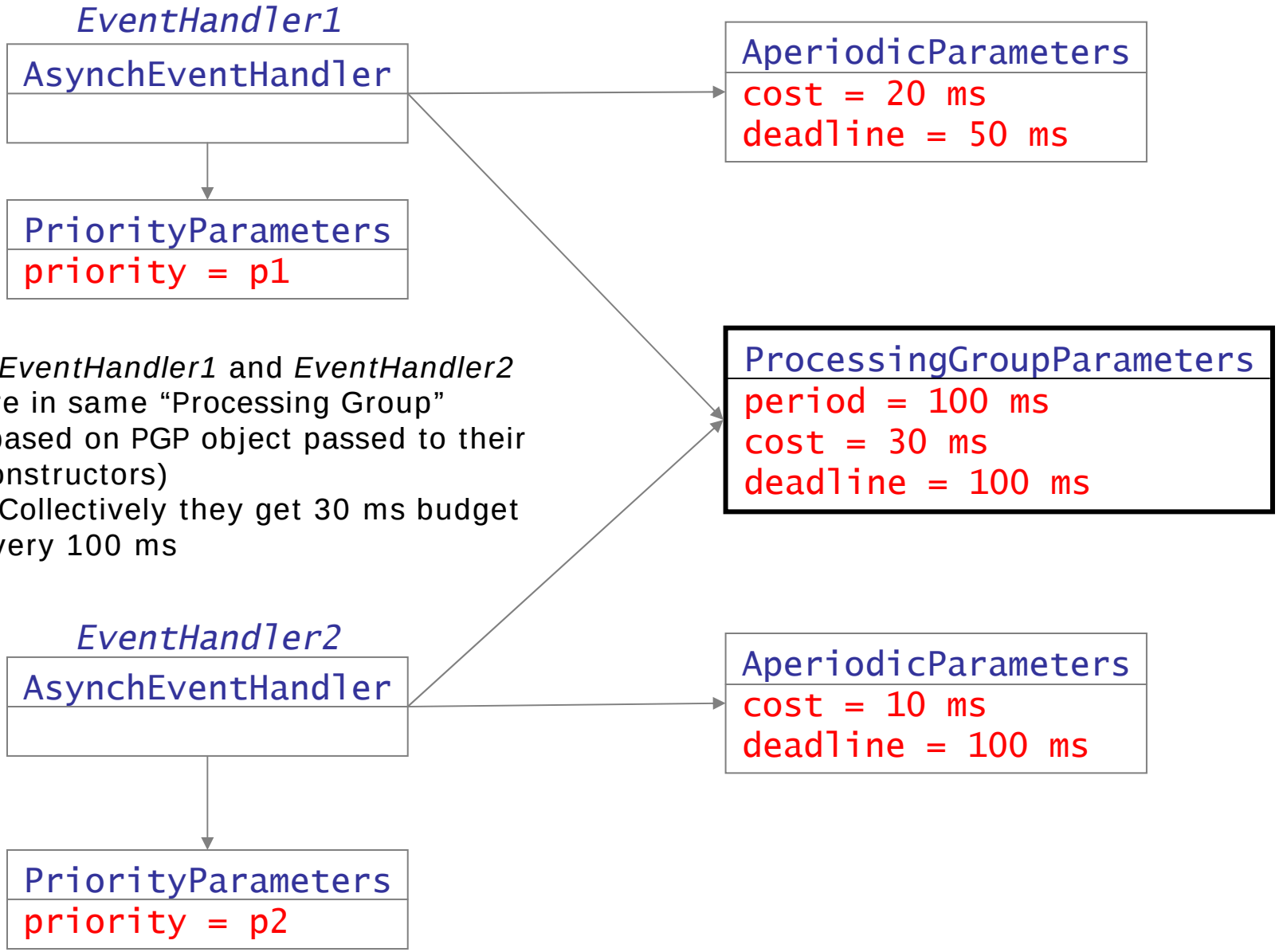
```
public ProcessingGroupParameters( HighResolutionTime start,
                                  RelativeTime       period,
                                  RelativeTime       cost,
                                  RelativeTime       deadline,
                                  AsynchEventHandler overrunHandler,
                                  AsynchEventHandler missHandler )
```

Methods

- Getters and setters for attributes (constructor parameters)

```
public boolean setIfFeasible ( RelativeTime period,
                               RelativeTime cost,
                               RelativeTime deadline )
```

ProcessingGroupParameters example



- *EventHandler1* and *EventHandler2* are in same "Processing Group" (based on PGP object passed to their constructors)
- Collectively they get 30 ms budget every 100 ms

Abstract Scheduler class

AdaCore

Constructor

```
protected Scheduler()
```

Methods (partial list)

- Set/get default scheduler (initially an instance of PriorityScheduler)

```
public static void setDefaultScheduler(Scheduler s)
public static Scheduler getDefaultScheduler()
```

- Get name of the scheduling policy

```
public abstract String getPolicyName()
```

- Feasibility analysis related methods

```
protected abstract boolean addToFeasibility(Schedulable s)
protected abstract boolean removeFromFeasibility(Schedulable s)
public abstract boolean isFeasible()
public abstract boolean setIfFeasible( Schedulable       s,
                                       ReleaseParameters r,
                                       MemoryParameters  m )
```

The PriorityScheduler class

Constructor

```
protected PriorityScheduler()
```

Methods (partial list)

- Methods to get min, max, and normal priority

```
public int getMinPriority()
public int getMaxPriority()
public int getNormPriority()
```

- Method to trigger execution of a Schedulable object as an AsyncEventHandler

```
public void fireSchedulable (Schedulable s)
```

- Static methods to get min, max, and normal priority for a Thread

```
public static int getMinPriority( Thread t )
public static int getMaxPriority( Thread t )
public static int getNormPriority( Thread t )
```

Notes

- Min, max, normal priority are methods (versus constants) since they are properties of run-time platform

Base scheduler – general properties

Represented by the object `PriorityScheduler.instance()`

Priority range

- Must support at least 28 distinct priority values, beyond the 10 for regular Java threads
- The 10 regular priority values have arbitrary mapping to native priorities
- In previous example, priority 30 should have been expressed more portably as `PriorityScheduler.instance().getMinPriority() + 20`

Preemptive, fixed priority, FIFO within priority

- Scheduler only changes a thread's priority for priority inversion avoidance
- Application may change a thread's priority dynamically
- Logical model is an array (indexed by priority) of ready queues
- "Run till blocked or preempted"
  - "Timeslice within highest priority" not permitted for base scheduler but may be supplied by implementation as additional scheduler

Base scheduler – queuing semantics

Several actions cause a thread* to go to the tail of the ready queue for its active priority

- A blocked thread becoming ready, via `notify()` / `notifyAll()`
- A sleeping thread becoming ready after delay expires
  - Including `waitForNextPeriod`
- A ready thread whose priority is changed via `PriorityParameters.setPriority()`
- A thread that performs a `yield()`

The RTSJ does not specify placement in the ready queue for a thread that is preempted or suffers loss of inherited priority

- Advises placement at head of queue
- Implementation must document its algorithm
- Non-determinism motivated by concerns over what RTOSes do in practice

** The term "thread" here includes schedulable objects*

Critique of scheduling mechanism

Advantages

- General, flexible, extensible framework
  - Various kinds of release characteristics reflecting common uses
  - “Hooks” for implementation-defined schedulers
  - Provision for different schedulers governing different threads
  - Allowance of dynamic replacement of schedulers
- On-line feasibility analysis with execution-time budget enforcement

Disadvantages

- Clumsy syntax
  - Constructors with positional notation and no default values
  - Retrieving min, max, norm priority values for base scheduler
- Design shows signs of “rush to publish”, and design bugs are difficult to correct without introducing incompatibilities
  - Missing some useful elements
  - Class hierarchy has some awkwardness
- Implementation-defined aspects of base scheduler
- Absence of other useful scheduling policies (round robin, EDF)
- Possible performance issues

Memory management in the RTSJ

Issue

- Garbage Collector may introduce delays that interfere with predictability and/or performance

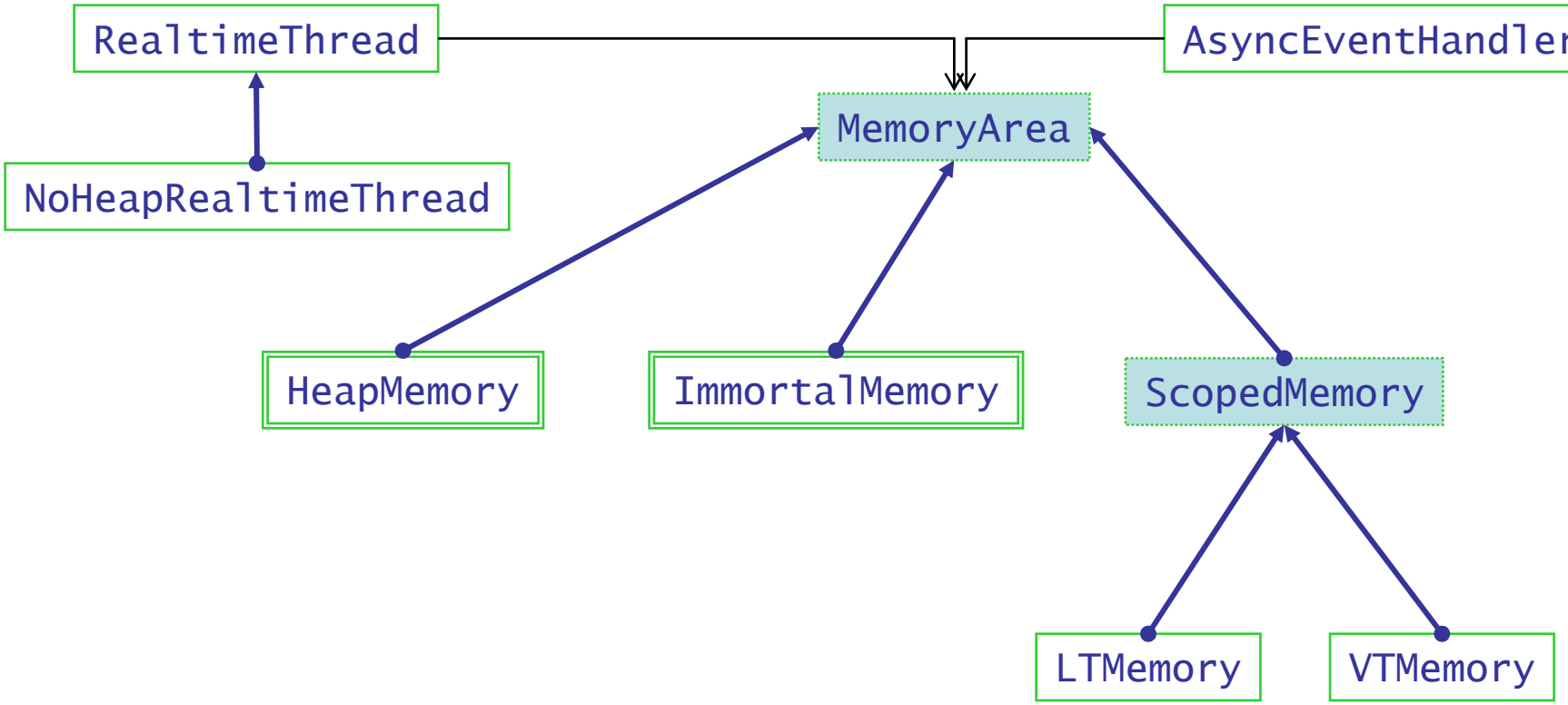
Goals

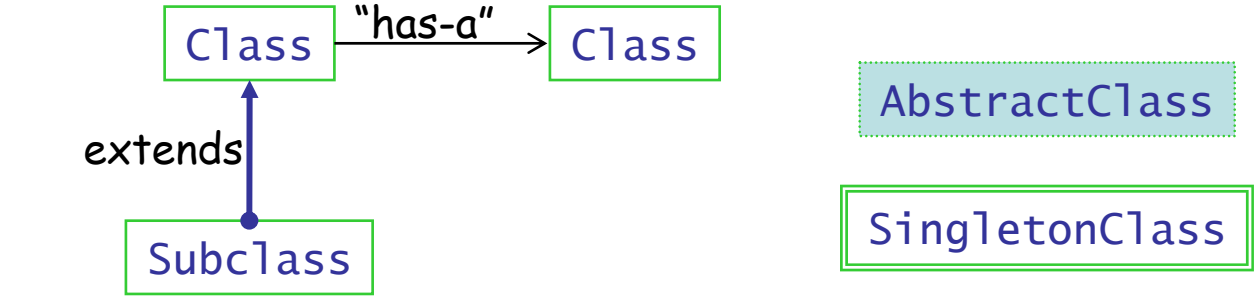
- Provide an efficient, general, predictable approach to memory management that can supplement or replace the regular GC
- Do not compromise Java safety (i.e., no explicit “free”)

Approach

- General concept of a “memory area” (context for object allocation)
  - Garbage-collected heap (singleton)
  - Non-GC’ed areas
    - Immortal memory (singleton)
    - Scoped memory (user-instantiable)
- Subclass of real-time threads that never reference the heap
  - They may preempt the GC and, in general, do not suffer GC latency
  - They may reference immortal or scoped memory

Memory management classes (partial list)





System-defined memory areas

Regular Java heap

- Singleton class `HeapMemory`
- Single instance given by static method `HeapMemory.instance()`
- Subject to the implementation's Garbage Collection algorithm
- Default memory area for object allocation

"Immortal Memory"

- Singleton class `ImmortalMemory`
- Single instance given by static method `ImmortalMemory.instance()`
- Some kinds of objects always are allocated in immortal memory
  - Class objects
  - Objects created during static initialization
  - "Intern"ed `String` objects
- Objects are not relocated or deallocated
  - Released only at program termination
  - Whether/when finalizers are run is implementation dependent
- May point to heap and vice versa
  - Thus GC needs to be able to scan immortal memory

Basic concepts

Definition

- The *closure* (of a method or constructor `f()` invoked by a thread or schedulable object *t*) is the sequence of statements executed by *t* for `f()`, including those in all methods or constructors called directly or indirectly from `f`, until `f` returns
  - Excludes the statements in the `run()` method of threads / schedulable objects that are started from `f()`, since such statements are not executed by *t*
- "Closure" is not a formal term in the RTSJ

You can bind the closure of the `run()` method for a `Runnable` object *r* to a memory area `ma`

- Within a given thread or schedulable object: `ma.enter( r )`
  - This is a *synchronous* call of r.run()
- Across multiple schedulable objects: pass `ma` as constructor parameter
- All object allocations during the closure will use `ma`, unless overridden by an inner binding to a different memory area

You can also bind a single constructor invocation to a memory area `ma`

- `ma.newInstance( class )`
- `ma.newArray( class, length )`

Scoped memory - preliminaries

Motivation

- In Java, only primitive data go on the stack, but most objects allocated in a method are only used locally in that method
  - Arrays and class instances thus need to be garbage collected
- Allow user to specify that some objects are “stackable”?
  - Does not work for objects that are returned as method results
- “Escape analysis” to automatically store objects on stack when safe?
  - Useful as an optimization, but not sufficient since some objects might still need to go on heap

Approach

- Allow user to construct non-GC’ed reusable memory areas that will be used for allocations during execution of specific code regions (closures) and which then get reset (all objects are deallocated after closure completes)
- Prevent dangling references

Classes

- Abstract class `ScopedMemory`
- Subclass `LTMemory` requires implementation to allocate / initialize with a linear time bound (based on object size)
- Subclass `VTMemory` allows “varying” (i.e. non-linear) time bound

Scoped memory - semantics

Basic usage similar to heap and immortal

- May be shared among / bound to several schedulable objects by being passed to their constructors
  - Used for allocations during the closure of the threads' `run()` methods
- May be used locally by a given schedulable object
  - Invoking `sm.enter(r)` activates the scoped memory `sm` for `Runnable` *r*'s `run` method and synchronously invokes `r.run()`
  - Used for allocations during the closure of `r.run()`

Distinctive features of scoped memory

- User creates scoped memory areas with specified sizes
- Each scoped memory area has an associated "backing memory" in which the allocated objects are stored
- A simple reference count scheme (number of referencing schedulable objects) establishes when a scoped memory area can be reset
  - Objects finalized, area "emptied" when last schedulable object bound to the area terminates, or when outermost enter returns
  - Reference count applies to memory area as a whole, not to individual objects
- Reference assignment is restricted, to prevent dangling references

AdaCore

The MemoryArea class (partial contents)

```
public abstract class MemoryArea{
  protected MemoryArea( long sizeInBytes );

  protected MemoryArea( long sizeInBytes, Runnable logic );
  // logic.run() is invoked whenever enter() is called on the constructee

  public void enter( Runnable r );
  // Binds this to the closure of r.run(), as the area used for allocations

  public synchronized Object newInstance(Class c)
    throws IllegalAccessException, InstantiationException, OutOfMemoryError;

  public synchronized Object newArray(Class c,int n)
    throws IllegalAccessException, InstantiationException, OutOfMemoryError;

  public static MemoryArea getMemoryArea(Object o);

  public long size();
  // current size in bytes (including unused space)

  public long memoryConsumed();
  // size (in bytes) of memory allocated to objects

  public long memoryRemaining();
  // approximate size (in bytes) of memory available for future allocation
}
```

Using a memory area

For (the closure of) a single allocation

```
ma.newInstance(someClass) or ma.newArray(someClass, n)
```

For the closure of a section of code: implement Runnable.run()

```
class MyRunnable implements Runnable{
  public void run(){ ... }
}
MyRunnable r = new MyRunnable();
ma.enter(r); // invokes r.run() synchronously
```

Alternative style using named object from an anonymous class

```
Runnable r = new Runnable(){
  public void run(){
    ... // code whose "new" objects are allocated in ma
  }
}
ma.enter(r);
```

Another style using anonymous object from an anonymous class

```
ma.enter(new Runnable(){
  public void run(){
    ... // code whose "new" objects are allocated in ma
  });
```

AdaCore

Example: a local ScopedMemory

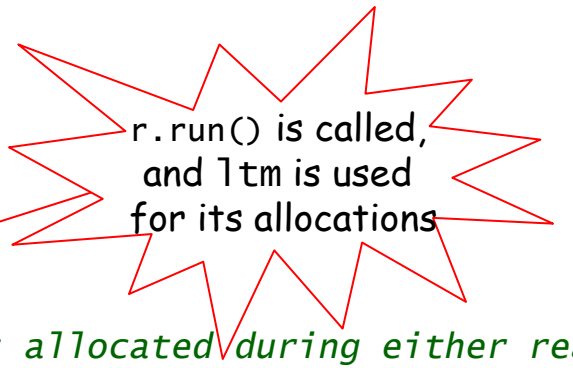
```
class InputPort{
   static double read(int j){ … } // reads double value from port j
}

class Sensor extends RealtimeThread{
  final Position ps;
  final LTMemory ltm = new LTMemory(10*1024, 10*1024); // Initial and max size = 10KB

  Sensor(Position p){
    super(new PriorityParameters(...), new PeriodicParameters(...) );
    ps = p;
  }

  public void run(){
    Runnable r = new Runnable(){
       public void run(){
          double[] arr = { InputPort.read(1), InputPort.read(2) };
          synchronized(ps){
             ps.x = arr[0];
             ps.y = arr[1];
          }
       }
    }
    while (true){
      ltm.enter(r);
      // arr, and any objects allocated during either read() are finalized,
      // and the area is emptied

      RealtimeThread.waitForNextPeriod();
    }
  }
}
```



Life cycle for a scoped memory area

`MemoryArea ma = new LTMemory(initSize, maxSize);`

- Allocates an LTMemory instance in the current memory area
- Creates ma's *backing memory* based on the size parameters
  - "Backing memory" is outside the Java realm (e.g. `malloc`'ed)

A *reference count* for the memory area instance is incremented when:

- ma is entered, on a Runnable r: `ma.enter(r)`
- A schedulable object with ma as memory area parameter is started

The reference count is decremented when:

- Control returns from `ma.enter(r)` (i.e., control returns from `r.run()`)
- A schedulable object which has ma as memory area parameter terminates

When the reference count goes from 1 to 0:

- Finalizers are run for all objects in the memory area
- The memory area is "emptied" (i.e., all objects therein are reclaimed, and the entire backing memory is again made available)

Note: the backing memory is not freed when the reference count goes to 0

Assignment rules

To \ From	Reference to Heap	Reference to Immortal	Reference to Scoped
Heap	Yes	Yes	No
Immortal	Yes	Yes	No
Scoped	Yes	Yes	Maybe (1)
Local Variable	Yes	Yes	Yes (2)

(1) Yes, if object containing the destination field is in same or inner (more recently entered) scoped memory; No, otherwise

(2) Always safe (no danger of dangling reference): object in scoped area not reclaimable until run() returns, which cannot occur before return from method in which the local variable is declared

Implementation may detect violations at compile time rather than throwing an exception (`IllegalAssignmentError`) at run time

In many potential application domains, such compile-time analysis will be important if the RTSJ is to be competitive with existing technologies

AdaCore

Example: a shared ScopedMemory (1)

```
class Sensor extends RealtimeThread{
  final Position ps;
  Sensor(Position p, MemoryArea ma){
    super(new PriorityParameters(...), new PeriodicParameters(...),
          null, null, ma, null, this);
    ps = p;
  }

  public void run(){
    // All allocations use the "backing memory" associated with ma
    while (true){
       double [] arr = { InputPort.read(1), InputPort.read(2) };
       synchronized(ps){ ps.x=arr[0]; ps.y=arr[1]; }
       try{
          this.waitForNextPeriodInterruptible();
       }
       catch (InterruptedException e){
          return;
       }
    }
  }
}
class Reporter extends RealtimeThread{...}
// analogous
```

Example: a shared ScopedMemory (2)

```
class Test{
  public static void main(String[] args){
    final Position p = new Position();
    final LTMemory ltm = new LTMemory(10*1024, 10*1024);
    // Initial size = max size = 10KB

    Sensor s = new Sensor(p, ltm);
    Reporter r = new Reporter(p, ltm);

    s.start();
    r.start();
    ...
    s.interrupt();  // terminate s at start of next period
    r.interrupt();  // terminate r at start of next period
  }

 // When s, r, and the main thread have terminated,
 // any objects allocated in ltm's backing memory
 // are finalized, and the area is cleared
}
```

Deallocation of the backing memory

The cleanup when a scoped memory area's reference count goes to 0 does not deallocate the backing memory

```
Runnable r = new Runnable(){
   public void run(){...}
}

void foo(){
   LTMemory m = new LTMemory( 100000, 100000 );
   m.enter(r);
   // Finalize all objects in m's backing memory, and reset to empty
}

while ( someCondition() ){
  foo();
}
```

- Each time `foo` is invoked, a new `MemoryArea` instance and 100KB backing memory space is allocated
- The objects are finalized and the area emptied when `enter` returns
  - But the backing memory is not reclaimed

In general, the backing memory can only be freed when its associated instance has been allocated in an outer scoped memory area, as part of the finalization of this instance

AdaCore

Idiom for scoped memory usage

The following style allows the same scoped memory area to be used repeatedly, with the area finalized / refreshed after each use

```
Runnable r =
     new Runnable(){
           public void run(){...}
         }

void foo(){
   LTMemory m = new LTMemory( 100000, 100000 );

   while ( someCondition() ){
      m.enter( r );
      // Finalize all objects in m's backing memory,
      // and reset the backing memory to empty
   }
}
```

Memory areas – further properties

Memory areas and schedulable objects

- The initial memory area for a schedulable object is defined either by a constructor parameter or by the creating thread's current memory area
- The memory areas currently active for a schedulable object form a stack

Restrictions on scoped memory allow simple "reference count" implementation ("single parent" rule)

- A scoped memory area may be shared or entered, but not both
- No nested enters on the same scoped memory area
  - If this were allowed, a permitted assignment could lead to dangling ref
  - To get around this restriction, method ma.executeInArea(Runnable) makes active area ma current

Regular Java threads may only use heap and immortal memory

- Constructing a Thread in a scoped memory area throws exception

Exceptions are thrown as usual

- Allocated in current memory area
- If attempt to propagate out of a scope, throw a ThrowBoundaryError exception (itself allocated in surrounding memory area)

Critique of memory area design

Advantages

- Scoped memory provides general, predictable, non-GC'ed mechanism
- `SizeEstimator` class allows implementation-independent definition for size of a memory area
- Safe (no explicit "free" / "unchecked deallocation")

Disadvantages

- Lots of objects need to be stored in immortal memory
- Need to take into account the time used in "emptying" a scoped memory area
- Reclamation of "backing memory" is implementation dependent
  - A method that allocates a scoped memory area may result in storage leakage if it is invoked in a loop
- Users need to be aware of memory management issues
  - Calling a library method when a scoped memory area is active may cause an assignment error (e.g., allocating an object and storing a reference in a static variable)
- Performance penalty for assignment checks, unless static analysis
- An exception propagating out of scoped memory areas will cause a cascade of `ThrowBoundaryError` exceptions
- Semantic complexity and subtleties

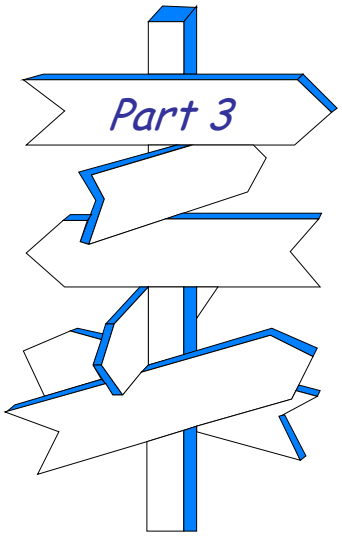
Part 3 – RTSJ (continued)

Synchronization

- Monitor control policies
- Priority Inheritance
- Priority Ceiling Emulation
- “Wait-free” queues

Asynchronous Event Handling

- The classes `Event` and `AsynchEventHandler`
- Example



Synchronization

Prevention of unbounded priority inversions

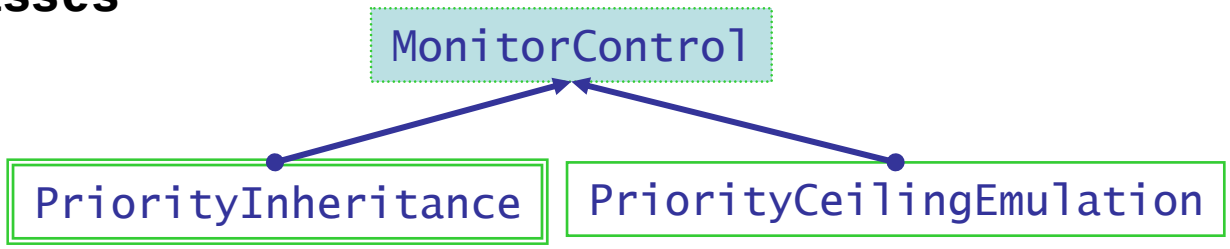
- General framework of "monitor control policy" (class `MonitorControlPolicy`)
  - User can define which policy governs which objects
  - Distinction between *active* and *base* priority
    - Base priority established on thread creation, may change as an effect of an explicit priority assignment
    - Active priority may change based on policies governing objects that a thread has locked (or is attempting to lock)
  - User may change an object's monitor control policy dynamically
- RTSJ defines `MonitorControlPolicy` subclasses for Priority Inheritance and Priority Ceiling Emulation
  - Specifies semantics for these policies (including their interaction) in the context of the base scheduler
  - Priority Inheritance is default policy, may be changed by user at system startup
  - Priority Ceiling Emulation support is optional

Minimization of GC-induced latency

- NoHeapRealtimeThread can preempt the GC at any time but may incur GC delay if it needs to share an object with a heap-using thread / schedulable obj
- "Wait-free queues" allow communication between a `NoHeapRealtimeThread` and a heap-using thread / schedulable object

Monitor control

Classes



Abstract class MonitorControl

- Methods to set policy globally or per-object

```
public static MonitorControl setMonitorControl(MonitorControl policy)
public static MonitorControl setMonitorControl(Object        obj,
                                                MonitorControl policy)
```

  - Return value is the old policy

- Methods to get policy globally or per-object

```
public static MonitorControl getMonitorControl()
public static MonitorControl getMonitorControl(Object obj)
```

Class PriorityInheritance

- Method to return the singleton instance

```
public static PriorityInheritance instance()
```

AdaCore

Base and active priority

Each thread* *t* has a *base priority* and an *active priority*

- Base priority is initially its priority at construction
- Updated immediately by several priority-setting methods

If *t* holds no locks, its base priority = active priority

If *t* holds one or more locks it has a set of *priority sources*

- Each lock defines one or more priority sources for the thread holding the lock
- *Active priority* is the maximum of *t*'s base priority and the priorities contributed from all its priority sources

Queuing effects for base scheduler

- Priority-setting method moves *t* to tail of queue at new (base) priority
- If the addition of a priority source increases *t*'s active priority, *t* is placed at the tail of the relevant queue at the new active priority
- If the removal of a priority source decreases *t*'s active priority, *t* should be placed at the head of the relevant queue at the new active priority (implementation advice, required if PCE is supported)

Priority sources may be added/removed either synchronously or asynchronously

---

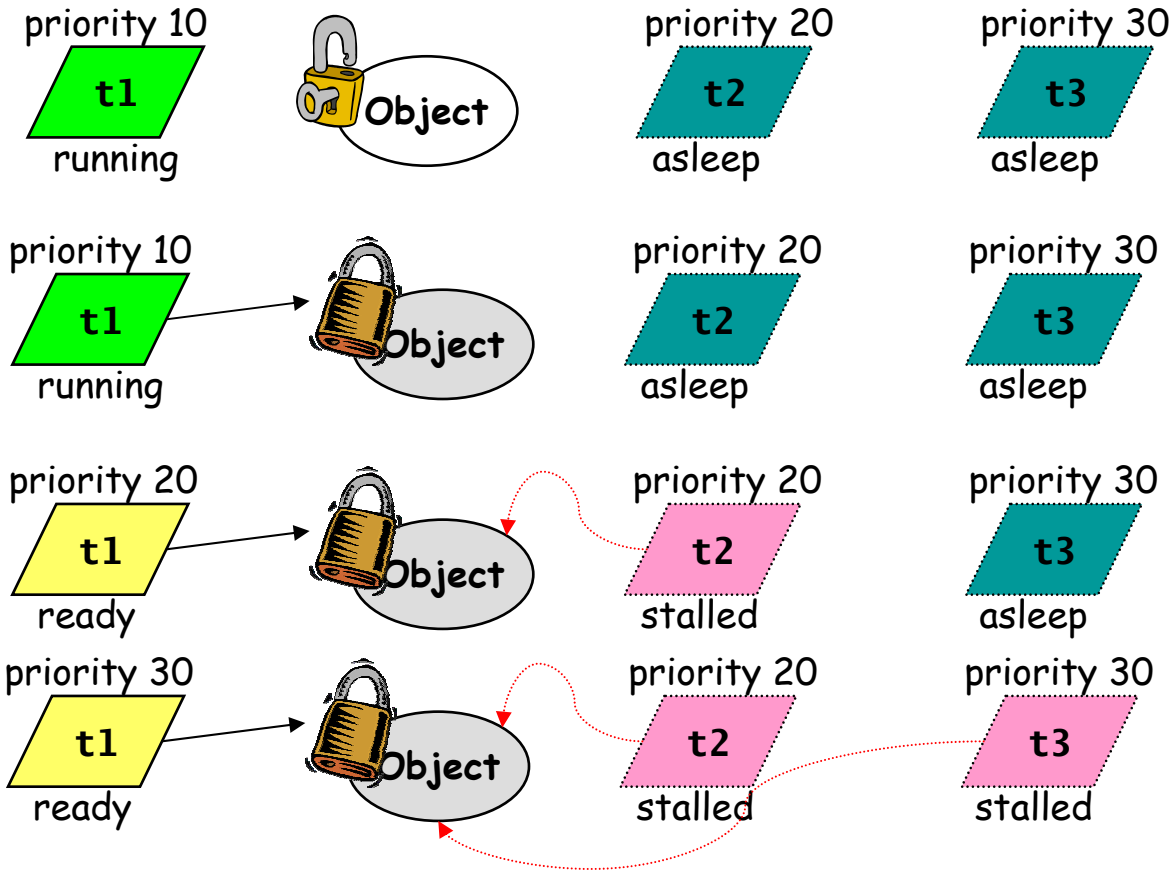
* In this section on Synchronization, "thread" includes schedulable objects

Priority Inheritance semantics

Priority sources for a thread t, from a Priority Inheritance lock held by t

- Each thread attempting to synchronize on the object governed by the lock
- Contributed priorities: active priority for each such thread

Example (object governed by Priority Inheritance policy)



Priority sources for t1 and the contributed priorities

{}

{}

{t2 (20)}

{t2 (20), t3 (30)}

Priority Ceiling Emulation

Class PriorityCeilingEmulation

- Informally, a thread synchronized on an object governed by a PCE policy has its active priority boosted to the policy's ceiling, while it holds the lock
- Unlike Ada, no requirement for non-blocking
- Immutable class, so need to `setMonitorControl(obj)` to change obj's ceiling

Method to create policy object / return existing object with given ceiling

```
public static PriorityCeilingEmulation instance( int ceiling )
```

- More efficient than constructor

Method to return ceiling for this policy object

```
public int getCeiling()
```

Method to return max ceiling supported by base scheduler

```
public static int getMaxCeiling()
```

Monitor control example

```
class Position{
  private double x, y;
  Position(double x, double y, int ceiling){
    this.x=x; this.y=y;
    MonitorControl.setMonitorControl(this, PriorityCeilingEmulation.instance(ceiling));
  }
  synchronized void setXY(double x, double y){
    this.x=x; this.y=y;
  }
  synchronized double[] getXY() {
    return new double[2]{x, y};
  }
}
class Sensor extends RealtimeThread{
  Sensor(Position p, int priority){...}
  ...
}
class Reporter extends RealtimeThread{
  Reporter(Position p, int priority){...}
  ...
}
class Test{
  public static void main(String[] args){
    Position p = new Position(0.0, 0.0, 16);
    Sensor   s = new Sensor(p, 15);
    Reporter r = new Reporter(p, 15);
    s.start(); r.start();
    ...
  }
}
```

PCE ceiling check (1)

Basic principle

- A thread attempting to lock an object governed by a PCE policy must have a priority no higher than the ceiling (else throw `CeilingViolationException`)
- Without the ceiling check, an unbounded priority inversion could occur:

PCE ceiling check (2)

Thread status at ①

- t1 (priority 21) running; t2 (priority 24) and t3 (priority 23) asleep

Thread status at ②

- t1 locks obj, is now running at (active) priority 22
- t2 and t3 still asleep

Thread status at ③

- t2 awakens, preempts t1
- t2 is running, t1 is ready
- t3 awakens, is made ready

Thread status at ④

- t2 attempts to lock obj
- In the absence of a ceiling check, t2 will simply stall (wait for the lock)
- Now t3, the higher priority of the ready threads, is allowed to execute

Thread status at ⑤

- t3 runs to completion, giving t2 an unbounded priority inversion

Ceiling check issues

Semantics (for a thread `t` attempting to acquire, or already holding, a PCE lock on `obj`)

- Which priority should be used: active? base? something else?
- When should the check be made?
  - When t attempts to synchronize on obj, whether obj is locked or not
  - When t attempts to synchronize on obj, but only when obj is not locked
  - When t performs a nested synchronization on obj
  - When t incurs a priority change while it is holding the lock on obj
    - Priority inheritance
    - Ceiling boost from nested synchronization on other PCE-governed object
    - Asynchronous priority-setting methods
    - Synchronous priority-setting methods
  - When t, after an obj.wait(), attempts to reacquire the lock on obj

Design goals that affect answers to these questions

- Avoidance of unbounded priority inversions
- Implementability (e.g. on platforms where the OS handles priority inheritance automatically)
- Minimization of likelihood of propagating an exception out of synchronized code
- Retention of PCE benefits (deadlock avoidance, no “chained blocking”)

PCE in RTSJ 1.0.1(a)

Semantics (for a thread t attempting to acquire, or already holding, a PCE lock on obj)

- Which priority should be used: active? base? something else? *Active*
- When should the check be made?
  - When t attempts to synchronize on obj, whether obj is locked or not ☑
  - When t attempts to synchronize on obj, but only when obj is not locked ☒
  - When t performs a nested synchronization on obj ☒
  - When t incurs a priority change while it is holding the lock on obj
    - Priority inheritance ☑ (exception thrown in thread causing the inheritance)
    - Ceiling boost from nested synchronization on other PCE-governed object ☒
    - Asynchronous priority-setting methods ☑ (exception thrown in thread attempting to perform the priority change)
    - Synchronous priority-setting methods ☑
  - When t, after an obj.wait(), attempts to reacquire the lock on obj ☑

Main idea

- "The active priority of a thread ... attempting to synchronize on, or already synchronized on, a target object governed by a PriorityCeilingEmulation policy, must not exceed that policy's ceiling"
- Motivation was to perform ceiling check (to avoid priority inversion) and also to prevent exception from propagating out of synchronized code

Critique of RTSJ 1.0.1(a) PCE semantics

Unnecessarily restrictive

- Need to prevent ceiling violation on first acquiring the PCE lock, but after the lock is held a ceiling violation does not risk unbounded priority inversion
- Restriction is methodological: priority boost may result in ceiling violation if the thread does a nested synchronization on another PCE-governed object

Implementation complexity

- A thread (priority p) that attempts to acquire a PI lock suffers a ceiling violation exception if the thread owning the lock also holds a PCE lock with ceiling < p
- Requires implementation to have a hook into the platform's priority inheritance mechanism
- May crash the VM if a system thread is the one attempting the synchronization

Confusing semantics

- Can a thread do nested synchronizations on objects governed by PCE locks with increasing ceilings? Intention was "yes", but rules implied "no"

Does not prevent propagation of exceptions out of synchronized code

- See next slide

Which priority for ceiling check?

Use of active priority causes problems, due to interaction between Priority Inheritance and Priority Ceiling Emulation

- A thread can inherit a high priority p if it holds a PI lock on an object, and then suffer a ceiling violation if it attempts to acquire a PCE lock with ceiling c<p
  - Programming/design error for making ceiling too low
- If this exception is not anticipated, it gets propagated out of synchronized code, leaving the PI-governed object in an inconsistent state

Thread t1 with base priority 10

```
① synchronized(objPI){
      ...
③     synchronized(objPCE15){
          ...
      }
      ...
  }
```

Thread t2 with base priority 20

```
      ...
② synchronized(objPI){
      ...
  }
  ...
```

① t1 (priority 10) locks PI-governed object

② t2 (priority 20) tries to lock objPI, t1's active priority boosted to 20

③ t1 suffers ceiling violation on trying to lock object with ceiling 15, synchronized statement for objPI is abruptly terminated

PCE in RTSJ 1.0.1(b)

Semantics* (for a thread t attempting to acquire, or already holding, a PCE lock on obj)

- Which priority should be used: active? base? something else? *Something else*
  - Max of base priority and ceilings of all PCE locks currently held
- When should the check be made?
  - When and only when t attempts to lock obj, whether obj is locked or not
- In order to avoid unbounded priority inversions, a PCE lock has priority inheritance semantics (when high active priority thread attempts to acquire it)

Priority sources for a thread t, from a Priority Ceiling Emulation lock (with ceiling c) held by t

Source	Contributed Priority
Each PCE-governed object locked by *t*	Max ceiling of all such PCE locks
Each thread trying to synchronize on a PCE-governed object locked by *t*	Max active priority of all such threads

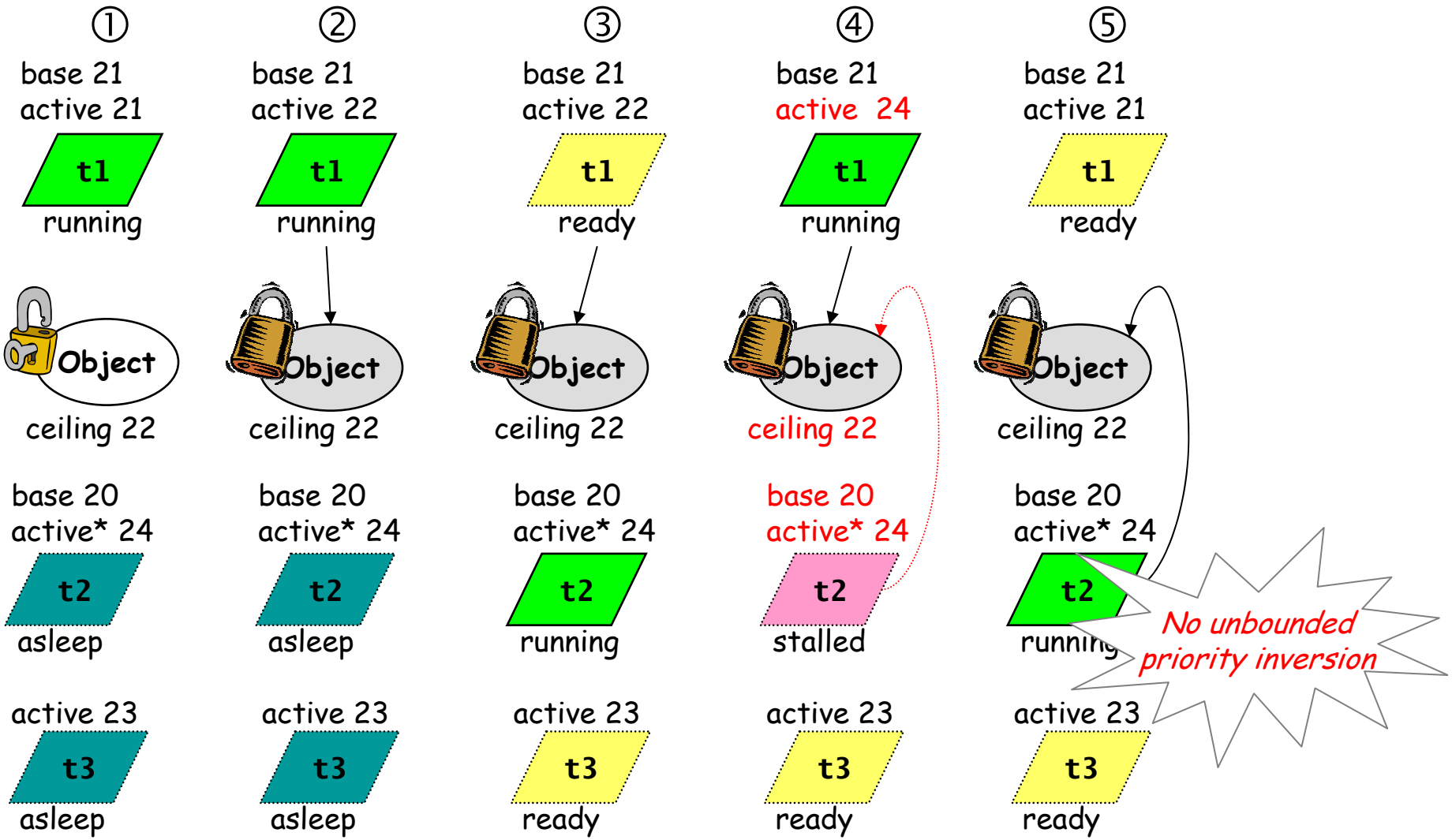
Ceiling that programmer needs to assign to an object obj

- Maximum of: (1) highest base priority for any thread t that can lock obj, and (2) max ceiling of any object locked by t when it is attempting to lock obj

* Main ideas credited to A. Wellings and A. Burns

PCE in RTSJ 1.0.1(b) – Example (1)

Ceiling check avoids unbounded priority inversion



* Assume from Priority Inheritance, not from ceiling on outer object

PCE in RTSJ 1.0.1(b) – Example (2)

Thread status at ①

- t1 (priority 21) running; t2 (priority 24) and t3 (priority 23) asleep

Thread status at ②

- t1 locks obj, is now running at (active) priority 22
- t2 and t3 still asleep

Thread status at ③

- t2 awakens, preempts t1
- t2 is running, t1 is ready
- t3 awakens, is made ready

Thread status at ④

- t2 attempts to lock obj
- Its base priority does not exceed ceiling, so t1 inherits t2's active priority and continues to run, while t2 is stalled
- t3 is ready threads, but cannot preempt t1 (no priority inversion for t2)

Thread status at ⑤

- t1 exits synchronized code, and t2 runs and locks obj

Non-blocking Priority Ceiling Emulation?

RTSJ does not prohibit synchronized code from blocking

- Thus queues / mutexes are needed to implement lock management

A more restricted form of PCE was considered for inclusion

- Throws an exception if a thread blocks while holding a lock

Advantages of non-blocking PCE

- More efficient implementation (no need for queues)
- Good model for interrupt handlers
- Interoperability with Ada protected objects

For simplicity, non-blocking PCE was omitted

- For example, would need to prevent assigning a non-blocking PCE lock to an object whose synchronized code can block
  - Otherwise exception will propagate out of synchronized code with the object in an inconsistent state

Implementation can add the policy but:

- Still needs to implement queue-based PCE
- Needs to ensure that the policy can only be assigned to objects that permit it

Immediacy of priority / policy change

Priority setting methods have immediate effect on base priority

- `pp.setPriority(prio)` for a `PriorityParameters` object `pp`
- `t.setSchedulingParameters(pp)` for a schedulable object `t`, where `pp` is a `PriorityParameters` object
- `t.setPriority(prio)` for thread / schedulable object `t`, where `prio` is an `int`

Base priority immediately affects active priority

- Alternative (deferring while thread holds locks) had unacceptable latency

Global policy change is immediate

- `public static MonitorControl setMonitorControl(MonitorControl policy)`
- Affects all objects subsequently constructed (no effect on existing objects)

Per-object policy change is immediate

- `public static MonitorControl setMonitorControl(Object obj, MonitorControl policy)`
- Invoking thread / schedulable object must hold lock on `obj`
- No special processing for threads /schedulable objects in `obj`'s wait set or stalled on the lock
  - E.g., no ceiling check if new policy has lower ceiling than old policy

Synchronization and NoHeapRealtimeThread (1)

Problem with priority inheritance if a NoHeapRealtimeThread t1 and a heap-using thread or schedulable object t2 attempt to synchronize on an object obj in ImmortalMemory or ScopedMemory

- The heap-using entity t2 locks obj
- The Garbage Collector preempts t2
  - May be triggered by "heap getting low" event
- The NoHeapRealtimeThread t1 preempts the GC and tries to lock obj
- The heap-using entity t2 inherits the NoHeapRealtimeThread's priority p1 (higher than the GC) and runs
- If t2 allocates a heap object the GC's data structures will be corrupted

Possible approaches to avoiding this problem?

- Throw an exception in t1 when it attempts to lock obj (no priority inheritance by t2)
  - Hard to program if a NoHeapRealtimeThread can have an exception thrown just because it is synchronizing on a PI-governed object
- Defer the priority inheritance by t2 until the GC is in a safe state
  - May cause unbounded priority inversion for t1, by lower-priority NoHeapRealtimeThreads whose priority exceeds that of the GC

Synchronization and NoHeapRealtimeThread (2)

Better solution involves a conceptual lock on the heap

- Accesses to the heap (by a heap-using thread or schedulable object, or by the GC) are controlled by a Priority Inheritance-governed lock
- In the above example, the GC owns the lock when it is preempted by the NoHeapRealtimeThread t1
- The heap-using entity t2 inherits t1's priority p1 and runs until it attempts to use the heap
- When t2 tries to acquire the heap lock, it will be blocked and the GC will inherit t1's priority

Consequences of this solution

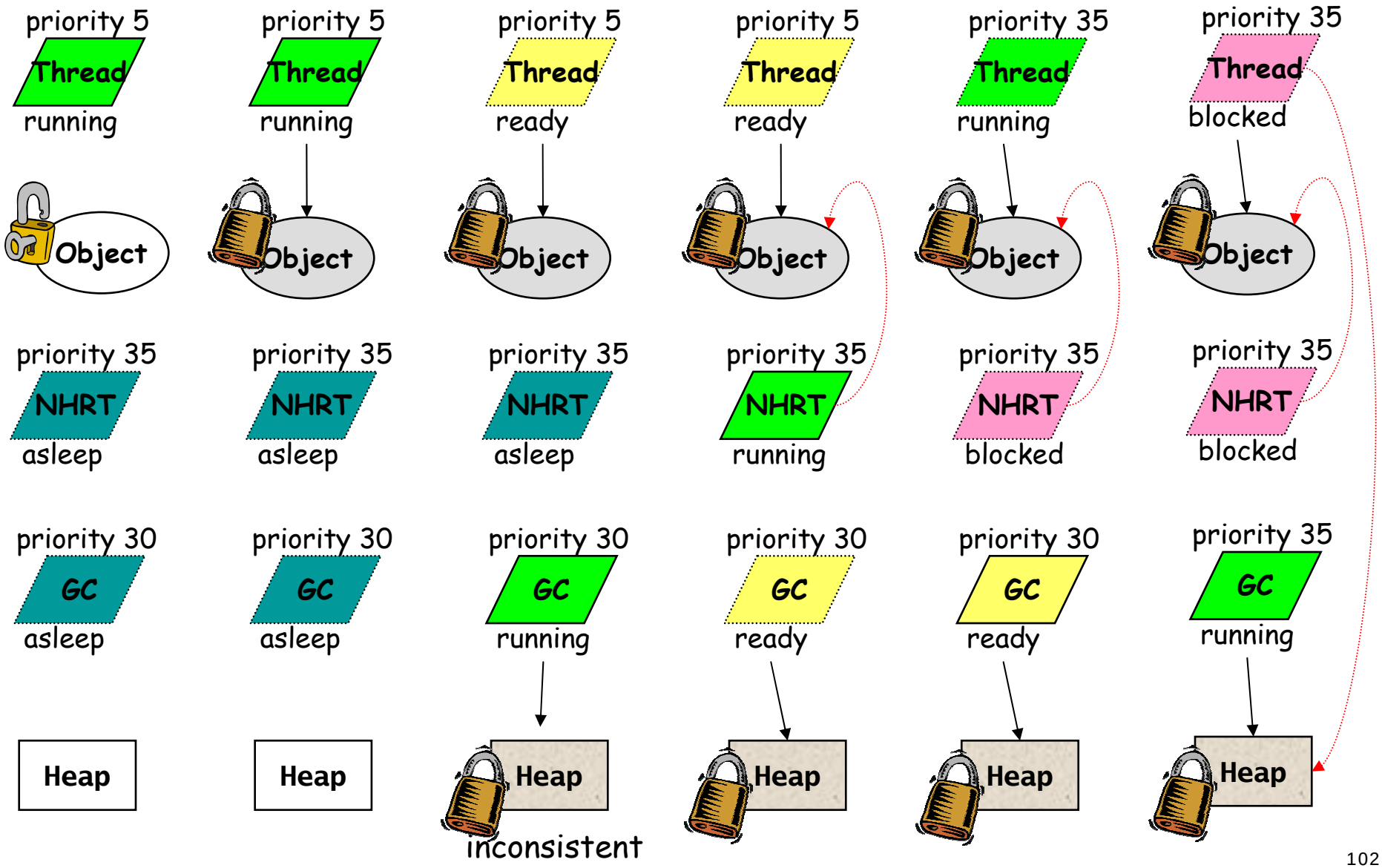
- Allows immediate priority inheritance, thus avoids unbounded priority inversions
- But it does introduce potential GC latency for a NoHeapRealtimeThread

Alternative programming solution

- If the GC latency is unacceptable, use a "Wait-Free queue" (versus synchronizing on an object) for communication between a NoHeap-RealtimeThread and a heap-using thread or schedulable object
  - Ensures that the NoHeapRealtimeThread does not block

Communication between no-heap, heap threads

GC may introduce latency for a NoHeapRealtimeThread



"Wait-free" queues (1)

Issue: how to share an object between a NoHeapRealtimeThread and a heap-using thread or schedulable object

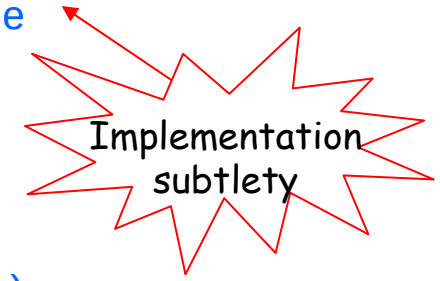
- Using synchronized code may lead to GC-induced priority inversions

Approach: two "wait-free" queues allow 1-writer multiple-reader or 1-reader multiple-writer communication

- Communication through bounded buffer of Objects, managed FIFO

WaitFreeWriteQueue class

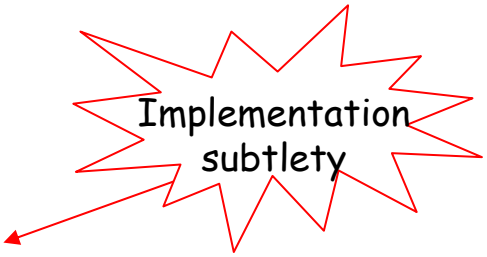
- Single writer (generally a NoHeapRealtimeThread)
- One or more readers (heap-using thread or schedulable object)
- `q.write(obj)` returns `boolean`
  - Non-synchronized, non-blocking
  - if `q` nonfull then insert `obj` and return `true`
  - if `q` full then return `false` (do not overwrite existing object)
  - In either event, awaken a waiting reader if there is one
- `q.read()` returns `Object`
  - Synchronized, potentially blocking
  - If `q` nonempty, return object least-recently written
  - If `q` empty, block (wakeup will be from eventual writer)



"Wait-free" queues (2)

WaitFreeReadQueue class

- Single reader (generally a NoHeapRealtimeThread)
- One or more writers (heap-using thread or schedulable object)
- Control if reader blocks on queue empty, unblocks on non-empty
- q.write(obj)
  - Synchronized, potentially blocking
  - if q nonfull then insert obj
  - if q full then block until there is room
  - Awaken blocked reader (if any) if q appropriately constructed



- q.read() returns Object
  - Non-synchronized, non-blocking
  - If q nonempty, return object least-recently written
  - If q empty, return null
- q.waitForData()
  - Non-synchronized, potentially blocking
  - If q nonempty, return immediately
  - If q empty, block (wakeup will be from eventual writer)
  - Style: `q.waitForData(); obj = q.read();`

Critique of synchronization approach

Advantages

- Highly flexible, with monitor control policies assignable per-object or globally
- Supports both priority inheritance and priority ceiling emulation
- Extensible (e.g. implementation may add new policy classes)
- Efficient approach to PCE policy instances (one per ceiling value)

Disadvantages

- No easy way to obtain optimization of non-blocking PCE
- Mixture of PI and PCE results in semantic complexity
  - Schedulability analysis needs to be researched
- Calculating appropriate ceiling for an object can be tricky, especially with asynchronous / immediate updates to policy and/or priority
  - Errors can result in exceptions propagated out of synchronized code
- Wait-free queues may leak immortal memory if not used carefully
  - User needs global knowledge of synchronization behavior
- Deprecated `WaitFreeDequeue` class shows signs of rushed design
- Optionality of `PriorityCeilingEmulation` interferes with portability

Asynchrony

An *asynchronous event* for a thread *t* affects *t* without *t*'s direct request, at an unpredictable point in *t*'s execution

- May be generated externally (e.g. from an interrupt or an OS signal) or internally (from another thread)
- A thread registers one or more `AsyncEventHandler` objects with an `AsyncEvent` object,
  - Scheduling parameters are supplied
- The handler is run (conceptually in an implementation-provided thread) in response to occurrence of an internal event or external "happening"

Handler can make its effect known to *t* in several ways

- Setting a variable polled by *t*
  - Example: input available
- Causing an asynchronous transfer of control in *t*
  - Example: time out and abandon computation

Goal

- Scale up to deal with large number of events and event handlers (tens of thousands), but with a smaller number of event handlers simultaneously active

Asynchrony basics

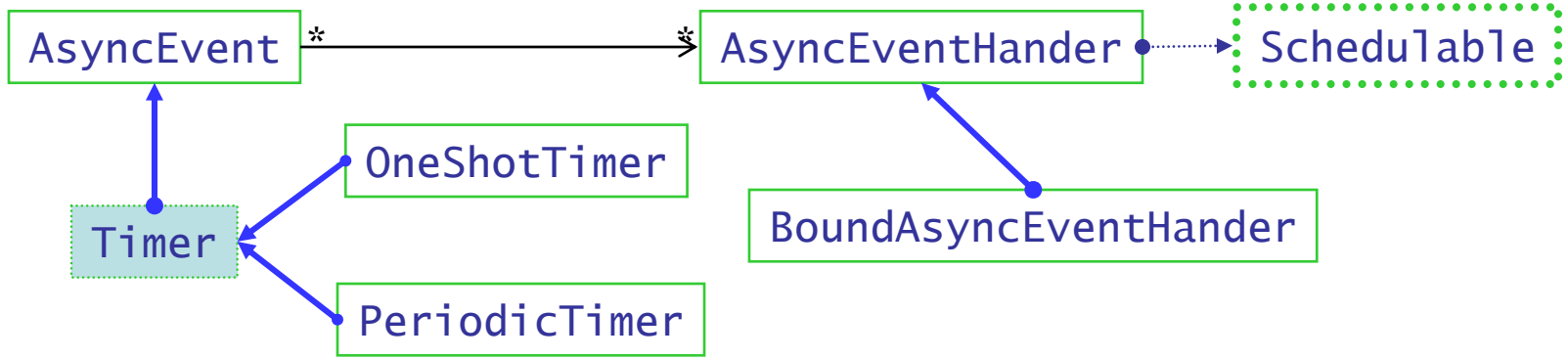
Asynchronous Event Handler ("AEH")

- Use for hardware interrupts or software "happenings"
- An AEH is a schedulable object but need not have a dedicated thread
- Override a method to implement the relevant event handling
- Associate one or more Asynch Event Handlers with an Asynch Event
  - Firing an AE → schedule associated AEHs
- Traditional Java "listener registration" model

Asynchronous Transfer of Control ("ATC")

- Use for timing out on a computation, aborting a thread
- Methodologically questionable, and complicated to implement
  - Conflict between desire for ATC to be immediate, and the need for certain code to execute completely
- Extends `t.interrupt()` to real-time threads, throwing an exception not only when `t` is blocked but also when `t` is executing *asynchronously interruptible* ("AI") code
  - Synchronized code, and methods lacking a special throws clause, are not AI

Asynchronous event handling classes (1)



The AsyncEvent class

- Public no-arg constructor
- Methods (partial list)

```
public void addHandler( AsyncEventHandler handler )
public void removeHandler(AsyncEventHandler handler )
public void setHandler( AsyncEventHandler handler )
public void bindTo( String happening )
public void unbindTo( String happening )
public void fire()
```

Asynchronous event handling classes (2)

The AsyncEventHandler class

- Constructors (partial set)
  - ```
    public AsyncEventHandler( SchedulingParameters scheduling,
                              ReleaseParameters    release,
                              MemoryParameters     memory,
                              MemoryArea           area,
                              boolean              nonheap,
                              Runnable             logic )
    ```
  - ```
    public AsyncEventHandler( SchedulingParameters scheduling,
                              ReleaseParameters    release,
                              MemoryParameters     memory,
                              MemoryArea           area,
                              boolean              nonheap )
    ```
- Methods (partial list)
  - `public void handleAsyncEvent()`
  - `protected int getPendingFireCount()`
  - `protected int getAndClearPendingFireCount()`
  - `protected int getAndDecrementPendingFireCount()`

  called from handleAsyncEvent

  - Getters and setters for “parameters” objects
  - Methods related to admission control / feasibility analysis

Asynchronous event handling semantics

AsyncEventHandlers are "registered" with an AsyncEvent by invoking `ae.addHandler(aeh)`

- Many-many relationship between events and handlers

An AsyncEvent ae is triggered by an invocation `ae.fire()`

- From a hardware interrupt handler or from a software thread
- No data passed
- Handler has event count in case of bursts

An AsyncEventHandler behaves conceptually like a thread (and thus has scheduling parameters etc)

- A `BoundAsyncEventHandler` has a dedicated thread

When an AsyncEvent ae is fired, the AEHs associated with ae are scheduled, and the `run()` method for each is executed asynchronously

- You do not override `run()` when defining an `AsyncEventHandler` subclass, but rather override `handleAsyncEvent()` which is called from `run()` and does the real work
- `run()` calls `handleAsyncEvent()` repeatedly as long as `fireCount > 0`

Asynchronous event handling example (1)

A hardware device writes sensor data to a pair of double values at locations 0x1000 and 0x1008

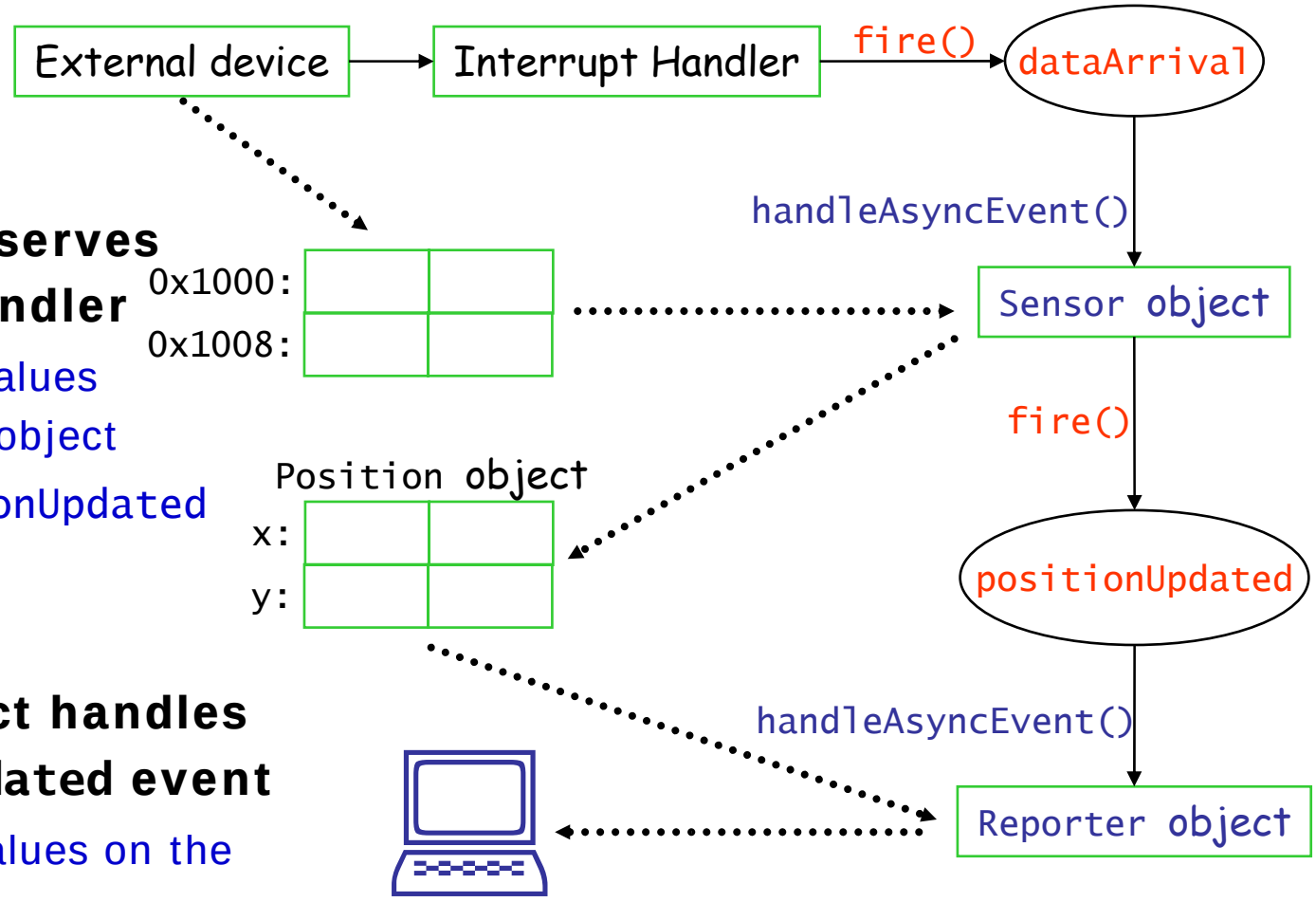
The interrupt handler fires a dataArrival event when the values are ready

A Sensor object serves as the event handler

- It copies these values into a Position object
- It fires a PositionUpdated event

A Reporter object handles the positionUpdated event

- It displays the values on the console



Asynchronous event handling example (2)

```
class AsynchEventExample{
  public static void main(String[] args){
    Position p = new Position();
    AsyncEvent dataArrival     = new AsyncEvent();
    AsyncEvent positionUpdated = new AsyncEvent();
    Reporter r = new Reporter(p, positionUpdated);
    Sensor s =   new Sensor(p, dataArrival, positionUpdated);
  }
}

class Reporter extends AsyncEventHandler{
  final Position pr;

  Reporter(Position p, AsyncEvent positionUpdated){
    pr = p;
    positionUpdated.addHandler(this);
  }

  public void handleAsyncEvent(){
     synchronized(pr){
       System.out.println(pr.x);
       System.out.println(pr.y);
     }
  }
}
```

Asynchronous event handling example (3)

```
class Sensor extends AsyncEventHandler{
  final RawMemoryFloatAccess rmfa;
  final Position ps;
  final AsyncEvent positionUpdated;

  Sensor(Position p, AsyncEvent dataArrival, AsyncEvent positionUpdated){
    rmfa =
      RawMemoryFloatAccess.createFloatAccess(
          "SRAM",  // type of memory
          0x1000,  // base address
          16);     // size (in bytes)
    ps = p;
    this.positionUpdated = positionUpdated;
    dataArrival.bindTo( "InputPort0Interrupt" );
    dataArrival.addHandler(this);
  }

  public void handleAsyncEvent(){
    synchronized(ps){
      ps.x = rmfa.getDouble(0);
      ps.y = rmfa.getDouble(8);
    }
    positionUpdated.fire();
  }
}
```

Part 4 – RTSJ (concluded) and related topics

Asynchronous Transfer of Control ("ATC")

- Basic concepts and alternative approaches
- Properties of asynchronous exceptions
- Examples: timeout, thread termination
- Comparison of RTSJ and Ada

Time and timers

- Classes
- Clocks and timers

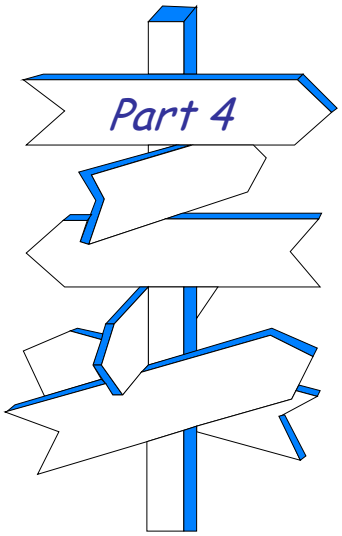
Low-level features

- "Raw storage"
- Physical memory

Other topics

- Comparison with Ada: process, technology
- The RTSJ in practice
- Safety-critical real-time Java
- RTSJ future development

Conclusions



Inter-thread communication

Synchronous

- Receiver ("target") suspends or polls at well-defined point
- Examples: rendezvous, `wait` / `notify`, "busy wait"

Asynchronous

- Target is notified "immediately" (no waiting or polling)
- Resumption model
  - Target executes response and then continues from the point of interruption
  - Example: interrupt handler
- Termination model
  - Target executes response but does not continue from the point of interruption

Asynchronous Transfer of Control

- A mechanism whereby a triggering thread (possibly an async event handler) can cause a target thread to branch unconditionally, without any explicit action from the target thread

Asynchronous Transfer of Control ("ATC")

Uses

- Timing out on a computation
- Awakening a suspended thread
- Aborting one iteration of a loop
- Aborting a thread

Benefits

- Predictable/bounded response time (avoids latency)

Problems

- Complicated to specify and implement
- Hard to program if control transfers may happen at unpredictable points
  - Triggering thread does not know what state the target thread is in when the ATC is initiated
  - Target thread must be coded carefully in presence of ATC
- May induce overhead even if not used
- *Conflict between desire for ATC to be immediate, and the need for certain code to execute completely*
  - Especially in synchronized code

ATC via asynchronous exception

Semantics

- Triggering thread throws asynchronous exception in target thread
- Control in target thread transfers to handler, or, if none, target thread terminates with unhandled exception

Advantages

- Intuitive conceptual model
- Gives target thread the ability to do cleanup

Potential issues

- Inconsistent state
- Unintended non-termination
- Unintended termination
- Nested ATCs / “competing” exceptions

ATC via abort

Semantics

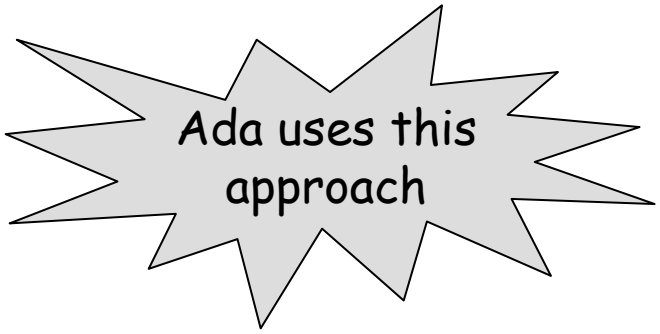
- Thread abort is the basic construct
- Code susceptible to ATC is conceptually an abortable thread
- No exception is thrown

Advantages

- Avoids exception-related anomalies

Potential issues

- Inconsistent state
- How to provide cleanup / prevent “resource leakage”



RTSJ's guiding principles for ATC

Methodological principles

- Susceptibility to ATC must be explicit in affected code
- Even if code allows ATC, in some sections ATC must be deferred; in particular, in synchronized code
- Code responding to an ATC does not return to the point where the ATC was initiated

Expressibility principles

- Need mechanism for explicitly triggering an ATC in a target thread
- Need to be able to trigger an ATC based on an external happening, a software event, or a timer
- Need to be able to abort a thread (safely)

Semantic principles

- If ATC modeled through exception handling, need to ensure that the exception is not caught by an unintended handler
- Nested ATCs (for example nested timers) must work properly

Pragmatic principles

- Need idioms for situations such as time-out and thread termination
- Programs that do not use ATC should not incur ATC-related cost

ATC semantics (1)

From the viewpoint of the target thread, ATC is modeled by exception handling

- Class `AsynchronouslyInterruptedException` extends `java.lang.InterruptedException`
  - We will use "AIE" as an abbreviation for `AsynchronouslyInterruptedException`
- An ATC is initiated by throwing an instance of AIE, as a result of some action by a triggering thread

Note: allowing a thread to arbitrarily throw an AIE at a target thread would have been problematic

- Target thread may be in code unprepared to catch the exception
  - If exception thrown too early or too late, it propagates and causes target thread to terminate with an unhandled exception
  - A "catch-all" handler may incorrectly handle the exception
  - Semantic mess to define effect of nested AIE propagation
- What if target thread holds a lock when the exception is thrown?
  - Abruptly terminating the synchronized code could lead to inconsistent data (if locks are released) or potential deadlock (if locks are not released)

ATC semantics (2)

From the viewpoint of the triggering thread, there are two ways to cause an ATC in a target thread

- Throw a "generic" AIE by invoking `t.interrupt()` where *t* is the target realtime thread
- Throw a specific AIE by invoking `aie.fire()`
  - The target thread must have indicated its readiness to "accept" `aie` being thrown asynchronously, else `aie.fire()` is ignored

ATC only occurs in code that explicitly permits it

- Static indication: method or constructor with a "`throws` AIE" clause

Some sections of code are *ATC-deferred*

- Synchronized code
- Any method or constructor lacking a "`throws` AIE" clause

Code that is not ATC-deferred is said to be *asynchronously interruptible* ("AI")

Asynchronous interruptibility is a static (lexical) property

- If non-AI code calls an AI method, the latter may be interrupted and the caller must handle the exception

ATC semantics (3)

How a target thread indicates susceptibility to interruption by a specific AIE instance (`aie`)

- Express the interruptible code as the `run()` method of an instance, *`i`*, of the `Interruptible` interface
- Invoke `aie.doInterruptible(i)`
  - This enables the thread to be interrupted by a triggering thread or `AsyncEventHandler` invoking `aie.fire()`
  - The AIE object referenced by `aie` must be accessible to both threads

Effect of `aie.doInterruptible(i)`

- Enable `aie`
- Invoke `i.run()` synchronously
- Disable `aie`

ATC semantics (4)

Semantics of `aie.fire()`

- If `aie` is enabled
  - If target thread is in an AI section, throw `aie` immediately
  - Otherwise keep `aie` as *pending* until execution next reaches an AI section and throw it then
    - "Next" may be either when invoking an AI method or constructor, or when returning (normally or abnormally) to such code
- If `aie` is disabled, `aie.fire()` is a no-op

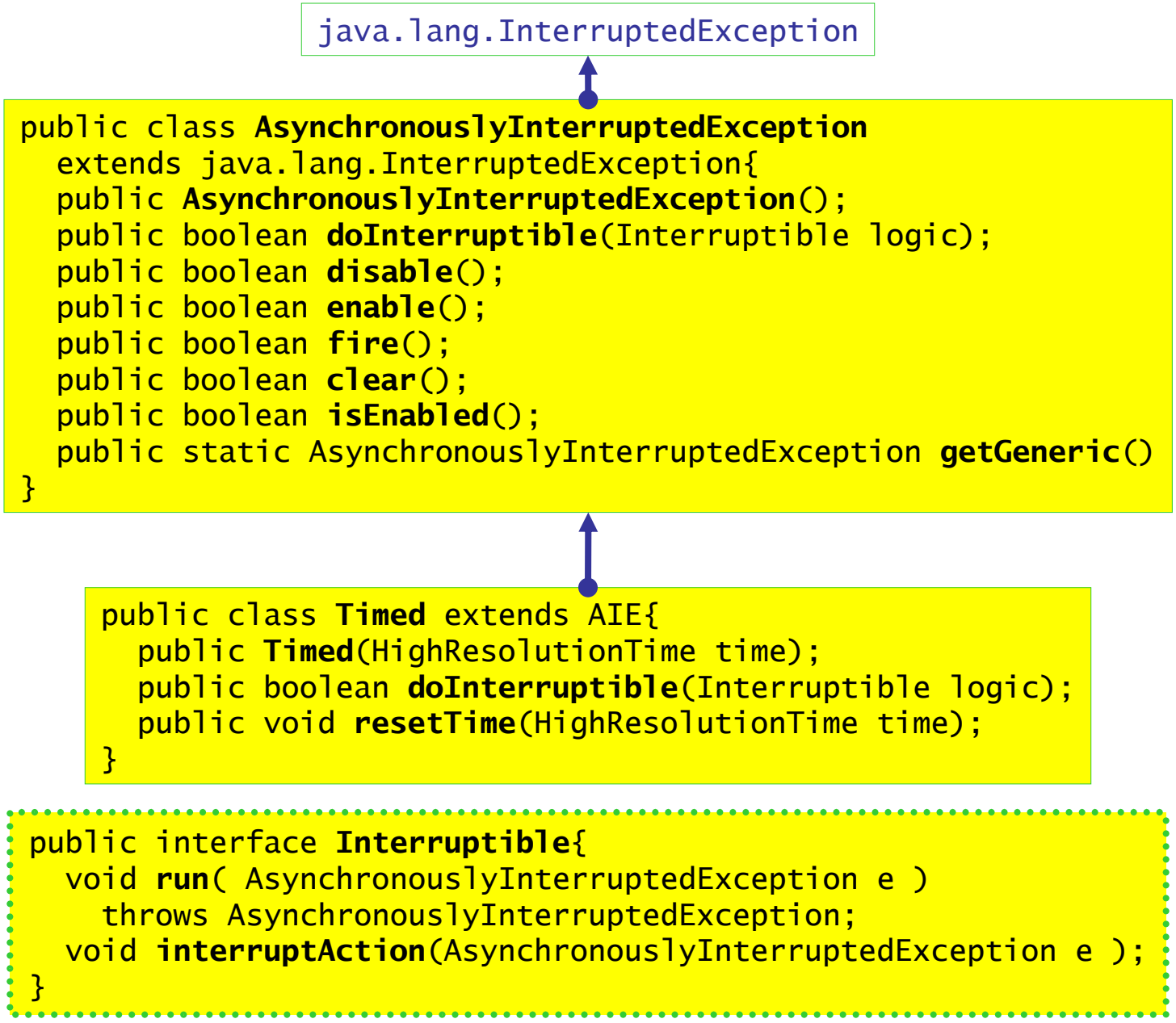
Semantics of `rt.interrupt()` for `RealtimeThread rt`

- Generalizes `Thread.interrupt()` to ATC
- A "generic" AIE is made pending on thread `rt`
- This AIE is thrown as soon as `rt` is in an AI section
  - Stays pending even if handled, unless/until a handler calls `clear()`

Nested AIE's

- Invocations of `doInterruptible()` may be nested
- The rules ensure that at most one AIE is pending
  - Precedence is given to the generic AIE and then to the named AIE directed at the shallower scope

ATC-related classes



ATC for timeout – basic idiom

```
new Timed(new RelativeTime( millis, nanos )).doInterruptible(
    new Interruptible(){
      public void run(AIE e) throws AIE{
        ... // code susceptible to asynch interruption
      }

      public void interruptAction(AIE e){
        ... // code executed if run interrupted
      }
    });
```

ATC example: time out

Example: use ATC to deliver a function result by successive approximation after a specified time interval

```
abstract class Func{
  abstract double f(double x)
    throws AIE;
  volatile double current;
   // assumes atomic
}

class MyFunc extends Func{
  double f(double x) throws AIE {
    current = ...;
    while(...){
      ... current = ...;
    }
    return current;
  }
}
```

```
class SuccessiveApproximation{
  static boolean finished;
  static double calc(Func   func,
                     double arg,
                     long   ms){
    double result = 0.0;
    new Timed( new RelativeTime(ms, 0) ).
       doInterruptible(
          new Interruptible(){
             public void run(AIE e) throws AIE{
                result   = func.f(arg);
                finished = true;
             }
             public void interruptAction(AIE e){
                result   = func.current;
                finished = false;
             }
          });
    return result;
  }
  public static void main(String[] args){
    MyFunc mf = new MyFunc();
    double answer = calc(mf, 100.0, 1000);
    // run mf.f(100.0) for at most 1 second
    System.out.println(answer);
    System.out.println("calc completed? " +
                       finished );
  }
}
```

Thread termination (1)

Goals

- Terminate a thread “immediately” and unconditionally but safely:
  - Don’t leave shared object in inconsistent state
  - Don’t deadlock

Original Java approach, and its problems

- The deprecated method `t.stop()` throws a `ThreadDeath` exception and unwinds the stack, releasing locks
  - This leaves shared objects inconsistent
  - A handler for `Exception` or `Throwable` will stop the termination
- The recently-deprecated and never-implemented method `t.destroy()` terminates the thread wherever it was, without throwing an exception
  - Thus any locks held by the thread are not released, which will deadlock any other thread that attempts to acquire any of them

Alternative solution in regular Java

- Use `t.interrupt()` and poll in t’s `run()` method to see when to return (“cooperative threadicide”)
- Problem: potential latency

Thread termination (2)

RTSJ Approach

- Use `t.interrupt()` and program `t` do its processing in an asynchronously interruptible method

Benefits of RTSJ Approach

- Safe, since the AIE is deferred while the target thread is in synchronized code
- Lower latency than with polling, when interrupted in an asynchronously interruptible section
- Author of target thread decides where the thread can be interrupted for termination

Drawbacks

- Termination not immediate (and indeed, if the thread never reaches an AI method then it will not be terminated)
- Notation can be heavy

General drawback to non-polling approaches

- Programming so that a program is robust in the presence of asynchronous thread termination is difficult

Thread termination example (1)

Thread that triggers the termination

```
class Executioner extends Thread{
  Thread victim;
  Executioner(Thread victim){ this.victim=victim; }
  public void run(){
    victim.start();
    ...
    if (...){victim.interrupt();}
    ...
   }
}
```

Style with polling

```
class PollingVictim extends Thread{
   public void run(){
     while (!interrupted()){
       ...
     }
     ... // pre-shutdown actions
   }
}
```

Thread termination example (2)

Style with ATC

```
class AbortableVictim extends RealtimeThread{
   private void body() throws AIE{
     ... // abortable here
   }
   public void run(){
     ... // not abortable here
     try{
       body(); // abortable here
     }
     catch( AIE e ){
       ... // response to abort
     }
     finally{
       ... // unconditional pre-shutdown actions
     }
   }
}
```

Comparison of ATC deferral in RTSJ and Ada

RTSJ uses lexical rules

- Synchronized code can call an AI method, and the latter can suffer an ATC
  - An AIE is propagated (synchronously) to caller
- ☺ Natural effect of exception-based semantics
- ☺ Reduces latency
- ☹ Complicates ensuring object consistency
- ☹ Issue with inline expansion

Ada uses dynamic rules

- If an operation is abort-deferred, then so is any subprogram that it invokes
- ☺ Preserves object consistency
- ☹ Increases latency

AdaCore

Critique of RTSJ's ATC model

Safety

- ☺ ATC is deferred in synchronized code
- ☺ ATC permission is per-method, via "throws AIE" clause
- ☹ ATC is possible in `finally` clauses
- ☹ `finally` clauses in AI code not executed when AIE propagates

Style / expressiveness

- ☺ Low-level mechanisms provide flexibility
- ☹ High-level idioms (e.g. `Timed` class) help, but basically the style is clumsy

Performance

- 😐 Latency depends on whether called methods are AI
- ☹ Expect distributed overhead, in absence of optimizations / restricted profile

Other

- ☹ Semantics of `finally` clauses interacts poorly with JSR 166 lock idioms

Comparison of ATC in Ada and RTSJ (1)

Semantics

- Ada abort-based model is simpler
- RTSJ has non-standard rules for AIE propagation, other special rules to avoid anomalies

Safety

- Ada has more reliable finalization behavior
  - `finally` clauses in RTSJ are asynchronously interruptible
  - `finally` clauses in RTSJ may be skipped in AIE propagation
- But RTSJ has finer-grained (per-method) control over ATC susceptibility, with "ATC deferred" as default
  - RTSJ: safer invocation of legacy code
  - Ada: subprogram's abort-deferral based on context
- Both ensure no ATC from synchronized / locked code, but at different levels of granularity

Comparison of ATC in Ada and RTSJ (2)

Style / expressiveness

- Ada is much more readable, with tailored syntax
- But RTSJ (actually regular Java) allows external unblocking of a suspended thread

Performance

- Latency is function of programming style
- Hard to avoid overhead, even if ATC not used

But ATC in general is a “last resort” feature

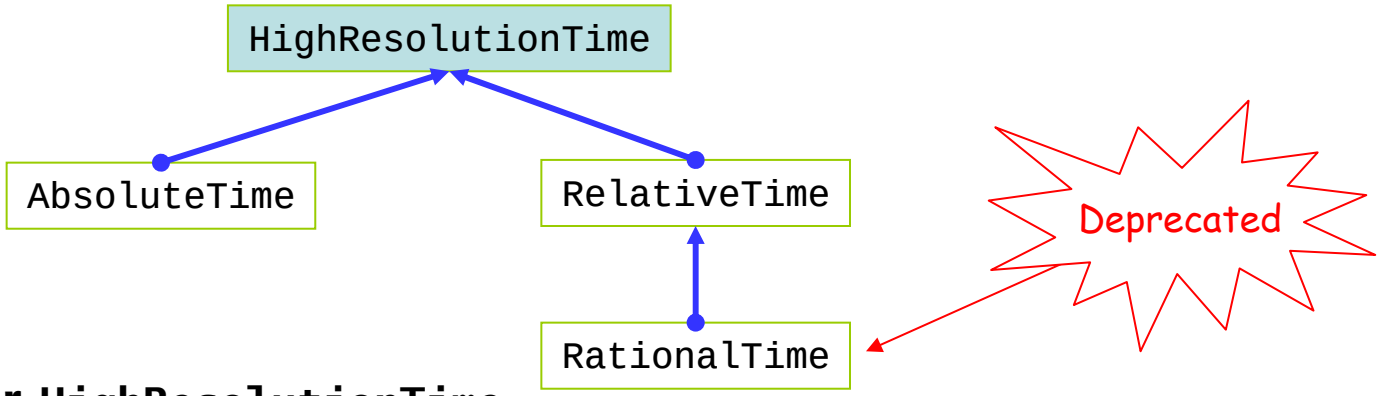
- Program is more readable, reliable, testable if all control transfers are synchronous
- Use ATC only for specialized circumstances
  - Program termination or “mode change”
  - Contexts where synchronous communication would incur unacceptably high latency

Time-related classes (1)

High-resolution time value

- [millis, nanos] where millis is 64 bits (long) and nanos is 32 bits (int)
- Normalized form has both components non-negative or both components non-positive, and nanos in range $-10^6 + 1$ .. $10^6 - 1$
- Always based on some Clock (by default the RealtimeClock)

Class hierarchy



Methods for `HighResolutionTime`

- Conversion to absolute or relative value ("in place" or return new object)
- Comparison (`compareTo`, `equals`)
- "Getters" and "setters"

Time-related classes (2)

AbsoluteTime class

- Various constructors, e.g.:
  - `AbsoluteTime(millis, nanos)` constructs value representing point that is [millis, nanos] after the origin of the epoch (00:00:00 GMT, 1/1/1970)
- Methods to associate a new `Clock` with an `AbsoluteTime` object
- Arithmetic operations, e.g.:
  - `absTime.add(millis, nanos) // return new object`
  - `absTime.add(millis, nanos, dest) // update dest`
  - `absTime.add(relTime) // add a RelativeTime, return new object`
  - `absTime.add(relTime, dest) // update by adding a RelativeTime`
- Methods to convert to a relative time based on an explicit clock

RelativeTime class

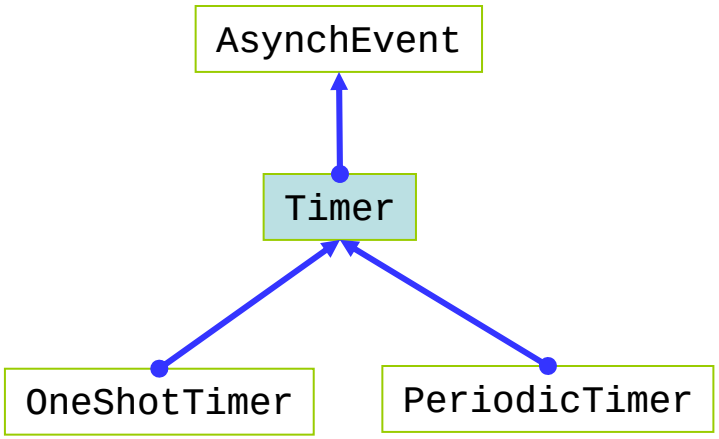
- Value represents millis and nanos relative to some `AbsoluteTime` for some `Clock`
- "Obvious" set of constructors, e.g. `RelativeTime(millis, nanos)`
- Methods to associate a new `Clock` with a `RelativeTime` object
- Arithmetic operations
- Methods to convert to an absolute time based on an explicit clock

Clocks and timers

The abstract Clock class

- "No-arg" constructor
- Methods
  - `public static Clock getRealtimeClock()`
    - "Singleton" object for a monotonic non-decreasing clock, not necessarily synchronized with external world
  - `public abstract AbsoluteTime getTime()`
  - `public abstract AbsoluteTime getTime(AbsoluteTime time)`
    - Returns the time parameter, into which the current time is copied
  - `public abstract void setResolution(RelativeTime resolution)`

The Timer class hierarchy



Timer classes

Abstract class Timer

- Constructor
  - ```
    protected Timer(
     HighResolutionTime time,     // When to fire event
     Clock              clock,    // Which clock
     AsyncEventHandler  handler ) // Which handler
    ```
- Methods
  - Methods to change timer's state (destroy, disable, enable, reschedule, start, stop)

OneShotTimer class

- Timed AsyncEvent that fires once (subject to enable / disable)
- Constructor is the same as for Timer, plus another one without an explicit Clock parameter

PeriodicTimer class

- AsyncEvent that fires periodically based on its parameters
- Constructors specify interval (RelativeTime) as well as start time, clock (optional), and handler
    ```

Example of periodic timer

```
public class Example{
   public void main( String[] args ){
      AsyncEventHandler aeh = new AsyncEventHandler(
         new PriorityParameters( 20 ),
         null,  // release parameters
         null,  // memory parameters
         null,  // memory area
         null,  // processing group parameters
         false, // heap
         new Runnable(){
            public void run(){ System.out.println( "Hello" ); }
      )

      PeriodicTimer pt =
       new PeriodicTimer(
         null,                          // activate at start
         new RelativeTime( 1000, 0 ), // every second
         aeh );                         // handler

      pt.start();                       // 1st period begins now
      Thread.sleep( 10000 );            // suspend for 10 sec
      pt.destroy();
   }
}
```

Critique of time-related classes

Advantages

- Satisfies requirements for nanosecond accessibility, high-resolution
- Supports relative and absolute time
- Accommodates monotonic clock
- Supports different kinds of timers

Disadvantages

- Class hierarchy is suspect
  - `RationalTime` deprecated
  - Occurrences of `ClassCastException` (e.g. for `compareTo` methods in `HighResolutionTime`)
- `Clock` is a weird class
  - Abstract class with a static method returning a singleton instance of one of its subclasses
  - Functionality of real-time clock would be better expressed as a singleton subclass `RealtimeClock` with a static `instance()` method
- Complicated state transition diagram for timers

Accessing "raw" or "physical" memory

RawMemoryAccess class

- Can construct a memory area of a given type (e.g. DMA) having a given base and size
- "Peek/poke" of byte, short, int, long (no references)
- Class RawMemoryFloatAccess includes float, double
- Constant BYTE_ORDER in class RealtimeSystem ⇨ endianness
- Example

```
RawMemoryAccess rm =
   new RawMemoryAccess( "RAM", 0x2000);
rm.setInt(0x1000, 0xABCDFFFF); // stored in bytes 0x1000 .. 0x1003
```

Physical Memory classes

- ImmortalPhysicalMemory, LTPhysicalMemory, VTPhysicalMemory
- Use for specialized hardware such as memory-mapped I/O
- Can contain arbitrary objects (not limited to primitive types as in RawMemoryAccess

Critique

- Captures some of the requirements for low-level programming, but (especially Raw Memory support) goes against the grain of Java

Comparison of Ada and RTSJ - the process

Ada

- Sponsored “top down” effort ⇨ ISO standard + Rationale
- Detailed audit trail (LSNs, AIs, etc.)
- Thorough review (ARG, WG9)
- Highly open process (public briefings, etc.)
- Product evolution based on ISO rules

RTSJ

- Focused “bottom up” volunteer effort ⇨ *de facto* standard
- Java Community Process requires not just the spec but also a Reference Implementation and Technology Compatibility Kit
- Audit trail comprises principally the group’s e-mail messages
- Review was principally internal in RTJEG
- Semi-open process
- Product evolution based on Sun’s JCP rules

Comparison of Ada and RTSJ: the technology

Ada

- ☺ Performance (classical stack-based language, queueless lock management)
- ☺ Conservatism (traditional static compile/bind/link)
- ☺ Well-defined semantics (queue placement)
- ☺ Cleaner / simpler approach to ATC
- ☺ Existence of good implementations now
- ☺ Allows but does not require OOP paradigm
- ☹ Market perception

RTSJ

- ☺ Flexibility (multiple schedulers, dynamic loading…)
- ☺ Functionality (admission control, …)
- ☹ Style may seem complicated to traditional Java programmers
  - Need to pay attention to memory management issues
  - OOP for real-time is not mainstream
- ☹ Performance questions

Ada's influence on Real-Time Java

Specific technical ideas

- Absolute delay (sleep() method taking an absolute time)
- Scheduling semantics for base scheduler
- Concept of "abort-deferred" regions of code
- Priority ceiling emulation
- Subsets for specialized application areas

Political lessons

- Customers want solutions, not technology
- Beware the culture clash
  - Real-time applications take a static approach to ensure predictability
  - All heap objects are allocated at system startup
  - OOP and garbage collection have not been popular

Challenges

- Sacrificing performance/flexibility for safety (an effect of Garbage Collection) has always been a hard sell to the real-time community

Ada and Real-Time Java: friends or foes?

Friends

- "The enemy of my enemy is my friend"
- Cross-fertilization of ideas beneficial to both
  - Many Ada concepts influenced RTSJ and Real-Time Core Extensions
    - Priority Ceiling, ATC, absolute delay, Ravenscar profile
  - RTSJ can serve as model for future Ada work in some areas
    - "On line" feasibility analysis, integrated support for real-time characteristics
- RTSJ-compliant JVM is feasible target for Ada

Foes

- Some organizations are looking at the RTSJ as an alternative to Ada

Peaceful coexistence

- Ada and RTSJ have somewhat different markets
  - Ada: traditional real-time, especially high-integrity
  - RTSJ: organization already committed to Java

AdaCore

J-Consortium's "core extensions"

Background

- Effort started at same time as the RTSJ, in part due to concerns over licensing issues associated with the Java Community Process
- Derived from Kelvin Nilsen's work on PERC VM and real-time GC

Objectives

- Extend Java platform with real-time capabilities, with services and performance similar to those of commercial RTOSes
- Serve as foundation for higher-level profiles

Functionality similar to RTSJ but from different perspective

- Concurrency model, priority inversion control, allocation contexts, asynchrony…
- Underlying decision: clear separation between "Baseline Java" and Core components (e.g., distinct class hierarchies, separate heaps)

Status

- Most recent design document was October 2000
- No implementations
- Web site www.j-consortium.com is defunct
- Main impact is likely to be indirect, via influence on other specs related to real-time Java and involvement by K. Nilsen

RTSJ implementations (1)

Reference Implementation from Timesys

- http://www.timesys.com

Sun Microsystems

- Project Mackinac
  - Production-quality implementation using Hot-Spot technology
  - Driven by customer requirements (power plant turbine control)
    - 16 ms period, 500 μs max latency/jitter, 3000 method calls per period
  - Integrated with OIS' OrbExpress
  - Led by Greg Bollella
- Project Golden Gate
  - Technology investigation between Sun and NASA Jet Propulsion Lab

Ovm (Open VVM)

- Open-Source framework for language run-time systems
- DARPA-sponsored; Purdue, SUNY Oswego, U. of Maryland, DL Tech
- Current thrust: produce JVM compliant with RTSJ
- http://www.ovmj.org/

RTSJ implementations (2)

Jamaica VM

- Implementation of RTSJ using F. Siebert's real-time GC algorithm
- http://www.aicas.com/jamaica.html
- HIDOORS project: http://www.hidoors.org/

High-Integrity Java

- European project for "defining and implementing a new High-Integrity Java for future networked real-time embedded systems"
- "Advance real-time systems implementation technologies to support the development of Architecturally Neutral, high-integrity Real-Time Systems (ANRTS)"
- 12 member organizations from European industry and academia
- http://www.hija.info

jRate (Java Real-Time Extension)

- Extends GNU GCJ front-end & run-time library, to support RTSJ
- http://www.cs.wustl.edu/~corsaro/jRate/

FLEX Project (MIT)

- Compiler infrastructure written in Java for Java
- http://www.flex-compiler.lcs.mit.edu/

Safety-critical real-time Java

Why consider?

- Sequential Java is secure, with implementation-independent semantics
- Lots of interest in Java technology in systems that have some safety-critical requirements

Some issues

- OOP
- Garbage Collection
- ATC

Approach

- Use RTSJ as basis, subset as required, and perhaps incorporate some ideas from J-Consortium's Core Extensions

Main properties

- No GC
- Limited API
- Compile-time checking where RTSJ has run-time checks

Status

- In progress, under The Open Group

Future plans for the RTSJ

Missing functionality from RTSJ 1.0.1(b)

- Support for aperiodic RealtimeThreads
  - Need RealtimeThread.waitForNextRelease
- Using blocking times in feasibility analysis
  - Add constructor for ReleaseParameters that includes max blocking time
- Querying the elapsed CPU time for a schedulable object
  - Add relevant method to Schedulable interface
- Allow programmer to associate an asynch event handler with the violation of an RTSJ resource limit
  - Currently some such errors only throw exceptions
- Flesh out time-related support
  - Query start time of Timers and RealtimeThreads
  - Allow reset of timeout for a Timed object
- Make scoped memory more flexible
  - Allow “weak references” between scoped areas

A new Java Spec Request (JSR) is currently in the planning stage

- RTSJ 1.1

Beyond that, maybe future JSRs for things like EDF, etc.

Conclusions

RTSJ has addressed the major technical issues concerning Java for real-time applications

- GC unpredictability ⇨ memory areas, `NoHeapRealtimeThread`
- Underspecified concurrency semantics ⇨ base scheduler
- Priority inversions ⇨ monitor control policies
- Issues with `stop()`, `destroy()` ⇨ ATC

RTSJ is attracting attention from the real-time research and user communities

Reasons that it may succeed

- Flexibility / generality / extensibility
- “Sexy” technology

Possible obstacles

- Design has “rough edges” (finalized before implementations)
- Performance?
- Style, some local to RTSJ and others due to Java limitations
- Problems with the Java thread model
- User conservatism about OOP

References (1)

Java thread model

- S. Oaks and H. Wong; *Java Threads (3rd Edition)*; O'Reilly; 2004; ISBN 0-596-00782-5
- D. Lea; *Concurrent Programming in Java (Second Edition)*; Addison-Wesley; 2000; ISBN 0-201-31009-0
- S.J. Hartley, *Concurrent Programming: the Java Programming Language*; Oxford University Press; 1998; ISBN 0-19-511315-2
- A. Holub; *Taming Java Threads*; Apress; 2000; ISBN 1-893115-10-0

Real-time Java: publications

- G. Bollella, J.Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, M. Turnbull, *The Real-Time Specification for Java*, Addison-Wesley; 2000; ISBN 0-201-70323-8
- J-Consortium Specification No. T1-00-01; *Real-Time Core Extensions for the Java Platform*; 2000
- P. Dibble; *Real-Time Java Platform Programming*; Prentice-Hall; 2002; ISBN 0130282618
- A. Wellings; *Concurrent and Real-Time Programming in Java*; John Wiley & Sons; 2004; ISBN 047084437X

References (2)

Real-time Java: web sites

- **www.rtj.org** (or **rtsj.dev.java.net**)
  - Official page for Real-Time for Java Experts Group
- **rtsj.pter.org**
  - Peter Dibble's RTSJ-related site
- **www.nist.gov/rt-java**
  - NIST Requirements