# Introduction to Real Time Java

Deniz Oğuz

Initial:22.06.2008
Update:05.12.2009

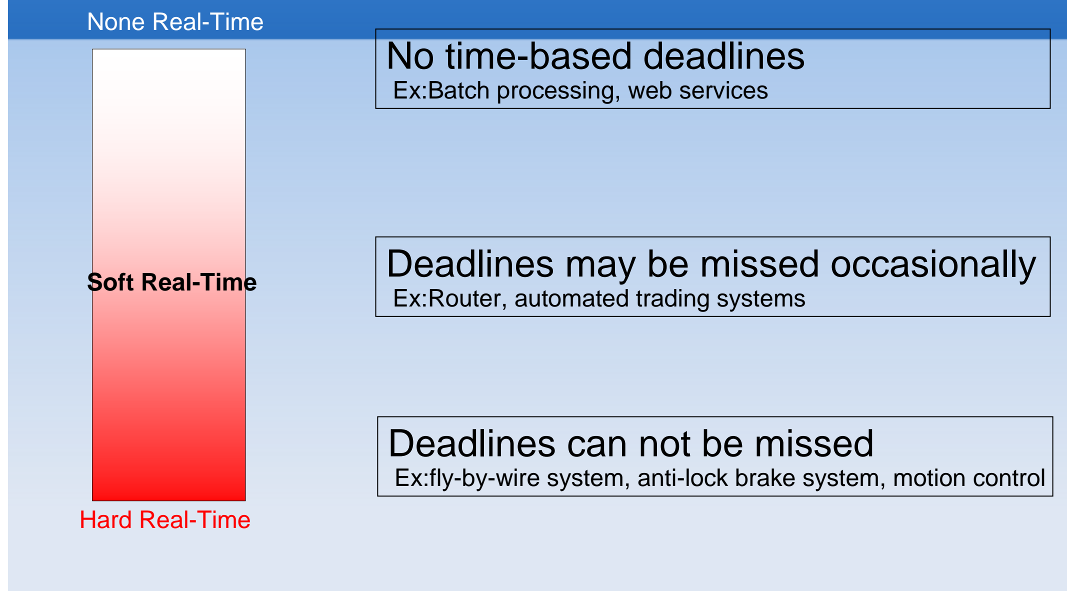
# Outline

- What is Real-time?
- What are the problems of Java SE?
- Real-Time Linux
- Features of RTSJ (Real Time Specification for Java), JSR 1, JSR 282
- IBM Websphere Real-Time
- SUN RTS

## What is Real-time?

- Real-time does not mean fast
  - Throughput and latency are important but does not enough
- Real-time is about determinism
  - Deadlines must be met
- Real-time is not just for embedded systems
  - Financial applications
  - Telecommunication applications
  - Network centric systems etc...

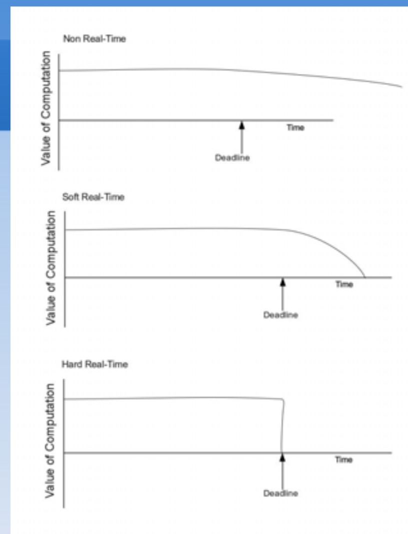
The first misconception is that real-time is minimal application latency and/or maximum data throughput. But actually real-time is about derminism. A system with latencies in the range of hundereds of milliseconds can also be real-time. Nor is real-time solely for embedded applications and systems; there is a growing need for enterprise determinism.
For example financial applications and telecommunication applications have very strict real-time requirements.
Determinism and throughput are usually inversely related. Determinism usually cost performance. A system that needs to bring its base-case and worst-case performances as close together as possible cannot use hints or heuristics and cannot rely on the ”80/20” rule. For example a quicksort ca takeO(n2) time so although a mergesort is slower than quicksort on average it is more suitable to real-time systems because it is predictable. The resulting software is typically slower than a software that is designed to optimize typical performance, but its worst-case performance may be an order of magnitude better than such a conventional design.




Saying a system is real-time does not mean that all parts of that system is real-time. It means there are certain tasks that should be performed in real-time.

# Hard/Soft real-time

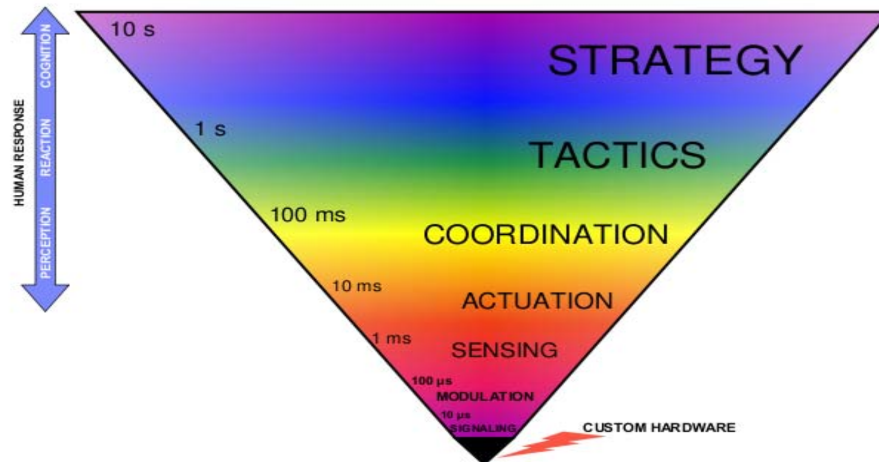
- Hard real-time
  - A late answer has no value
- Soft real-time
  - late answer has still a value but value of answer rapidly degrees




In a hard real-time system, a late answer is the same with no answer or wrong answer.
In a soft real-time system, a late answer is still useable but value of it decreases rapidly. In other words occasional dead-line misses are acceptable.
The severity of the consequence of missing a deadline has nothing to do with the definition of hard versus soft.

## Software Complexity vs. Time

Real-time Systems: Software Complexity vs. Time Domain




Taken From **IBM WebSphere Real Time: Providing predictable performance** by Michael S. Fulton, Java chief architect;Darren V. Hart, Real Time team lead; andGregory A. Porpora, IBM Software Group. December 2006

As the requirement for deterministic behavior moves from tightly coupled embedded tasks that control or monitor hardware subsystems to highly complex enterprise applications, software complexity and deterministic latency requirements become inversely proportional to solution needs and capabilities.

General real-time software usually deals with time measured in milliseconds. Most real-time systems fall into this range. These systems can be programmed with normal tools.

In practice, achieving response times below tens of microseconds requires a combination of custom hardware and software, possibly with no -- or a very thin -- operating-system layer.

Within this overall complexity, architects, systems designers and developers need an environment that can span a significant portion of the response
spectrum. This model must not only meet a broad bandwidth of performance and latency requirements, but also be scalable and relatively simple to use.

## Latency and Jitter

- Latency is the time between external event and system's response to that event
- Jitter is the variation in latency

Aim of a real-time system is to put a deterministic bound to latency not to minimize it. The goal is to make latency known, small enough, consistent and measureable quantity.

Jitter is the variation in latency.

# Hardware Architecture & OS

- Modern processors and operating systems are optimized for throughput
  - It makes excellent sense for most systems to trade a rare factor of 100 times slowdown for a performance doubling everywhere else

# Hardware & OS Cont.

- Worst-case scenario for an instruction
  - Instruction is not in the cache, processor must read it from memory
  - An address translation cache miss requires more memory access
  - Instruction might be in demand paged memory
  - Data may not be in cache
  - A large DMA may delay operations
  - SMI on X86 platforms
  - .......

# Worst Case Execution Cont.

**This example is taken from Real-Time Java Platform Programming by Peter C. Dibble**

| Event | Estimate Time (ns) |
|---|---|
| Execute Instruction | 10 |
| Instruction cache miss | 50 |
| Instruction ATC miss | 500 |
| Data cache miss | 100 |
| Dirty data cache write ATC miss | 500 |
| Data cache read ATC miss | 500 |
| Demand paging for instruction read (write dirty page) | 20000000 |
| Demand paging for dirty data cache write (read page) | 10000000 |
| Demand paging for data cache write (write dirty page) | 20000000 |
| Demand paing for data cache write (read page) | 10000000 |
| Demand paging for data cache read (write page) | 20000000 |
| Demand paging for data cache read (read page) | 10000000 |
| Interrupts | 100000 |
| One Big DMA | 10000000 |
| **Total** | **100101660** |

As you can see worst case execution time can be 1 million times of best execution time. There can be other factors not included in this example. For example SMI (System Management Interrupts) on X86 hardware may add extra delay.

## How to prevent worst case

- Disable paging for time critical code
- Use processor affinity and pin RT Threads to cpus
- Use tunable DMA
- Use large pages to reduce load on TLB
- Disable interrupts or give RT Threads higher priority than some interrupts
- On x86 architecture pay attention to SMI (System Management Interrupts)
  - Do not disable it completely , you may burn down your processor.

Even non rt windows and linux allows memory of a process to be locked in to memory. So at worst everything your program needs will be in memory.
Some DMA controllers can be tuned to use no more than a specific faction of memory bandwidth or get off the bus entirely when the processor is servicing interrupts
TLB is limited resource on processors. They are used to translate virtual addresses to physical addresses. Normally most operating systems uses 4K pages. For each page a TLB entry is required. You can reserve some number of large pages like 256Megabytes for your process.

# Processor Pinning

- Less context switching jitter
- Decreased cache miss
- Example (Linux and Sun RTS only):
  - -XX:RTSJBindRTTToProcessors=0,1
  - -XX:RTSJBindNHRTToProcessors=0,1
  - Alternatively you can create a cpu set named xx and use /dev/cpuset/xx

Pinning Your Threads to processors can help to increase determinism. Both solaris and linux supports processor pinning. Solaris also supports interrupt shielding. Not all linux distributions support ineterrupt shielding. Your accound should have sufficient privileges to use processor pinning.

As you can see you you can not select which thread works on which CPU. Your RealTime and NoHeapRealTime threads will use assigned cpus and all other applications and other JVM Threads (GC threads, JIT compiler thread etc.) will use other cpus in the system.

# Large Memory Pages

- Garbage collectors performs very bad if memory is swaped to disk
- For increased determinism use large memory pages and pin these pages to memory
- Example Linux and Sun's Java SE only:
  - echo shared_memory_in_bytes > /proc/sys/kernel/shmmax
  - echo number_of_large_pages > /proc/sys/vm/nr_hugepages
  - Start JVM using XX:+UseLargePages argument
  - Verify using cat /proc/meminfo | grep Huge

Refer following Sun article for large memory pages: Java Support for Large Memory Pages

Beginning with Java SE 5.0 there is a cross-platform flag for requesting large memory pages: **-XX:+UseLargePages** (on by default for Solaris, off by default for Windows and Linux). The goal of large page support is to optimize processor Translation-Lookaside Buffers.

A Translation-Lookaside Buffer (TLB) is a page translation cache that holds the most-recently used virtual-to-physical address translations. TLB is a scarce system resource. A TLB miss can be costly as the processor must then read from the hierarchical page table, which may require multiple memory accesses. By using bigger page size, a single TLB entry can represent larger memory range. There will be less pressure on TLB and memory-intensive applications may have better performance. Using large memory pages may negatively affect performance of system if it cause memory shortage for the rest of the system.

# RT-POSIX

- An extension to POSIX standard to address hard and soft real-time systems (POSIX 1003.1b)
  - Priority Inversion Control and Priority Inheritance
  - New schedulers
  - Asynchronous IO
  - Periodic, Aperiodic and Sporadic Threads
  - High Resolution Timers
  - RT File System
  - Some others ....

For further information refer to RT Posix standard

## RT Linux

- Fully preemptible kernel
- Threaded interrupt handlers for reduced latency
- High-resolution timers
- Priority inheritance
- Robust mutexes and rt-mutexes
- To install use

  ***sudo apt-get install linux-rt***

  in ubuntu. Select rt kernel at next system start-up

A standard Linux kernel provides soft RT behavior, and although there's no guaranteed upper bound on how long a higher-priority thread waits to preempt a lower-priority thread, the time can be roughly approximated as tens of milliseconds In RT Linux, almost every kernel activity is made preemptible, thereby reducing the time required for a lower-priority thread to be preempted and allow a higher-priority one to run. Remaining critical sections that cannot be preempted are short and perform deterministically. RT scheduling latencies have been improved by three orders of magnitude and can now be measured roughly in tens of microseconds.

Almost all interrupt handlers are converted to kernel threads that run in process context. Latency is lower and more deterministic because handlers become user-configurable, schedulable entities that can be preempted and prioritized just like any other process.

High-resolution time and timers provide increased resolution and accuracy. RT Java uses these features for high-resolution sleep and timed waits. Linux high-resolution timers are implemented with a high-precision, 64-bit data type. Unlike traditional Linux, where time and timers depend on the low-resolution system tick -- which limits the granularity of timer events -- RT Linux uses independently programmable high-resolution timer events that can be made to expire within microseconds of each other.
RT-Linux prevents priority Inversion by using priority inheritance. Following slides contains more information about this topic.

## Why Java?

- Software (including embedded software) becomes more complex and gets unmanageable with old practices and tools
  - Ex:Financial systems, Network Centric systems
- Single language, tools for real-time and non-real time parts
- Java Platform provides a more productive environment
- A large set of 3rd party libraries are available

Has a very big community

  - Large number of developers
  - Support from big companies like IBM, SUN, Oracle
- A lot safer when compared to low level languages

Java RTS real-time components and non-real-time components can coexist and share data on a single system. An RTS enabled JVM is fully compatible with Java SE giving developers unforeseen flexibility.

# Cons of Using C and Java Together

- Same functionality is codded twice
- High amount of integration problems
- Interfaces between C and Java is ugly and introduce overhead
- Communication via JNI can violate safe features of Java
- The JNI interface is inefficient
- Increased maintenance cost in the feature due to above problems

## What are the problems of Java SE?

- Dynamic Class Loading and Linking
- JIT (Just in Time Compiler)
- Thread Handling
- Garbage Collector
- No Raw Memory Access
- No support for real-time operations, like deadline miss handling, periodic scheduling, processor pinning etc.

Due to these constrain Java is not suitable for real-time applications. One alternative may be codding real-time part of the application with another language, like C, and code the remaining non-real-time parts using Java. This technique is widely used in the past.

## Dynamic Class Loading

- A Java-conformant JVM must delay loading a class until it's first referenced by a program
  - Early loading is not allowed so JVM can not do this for you at application startup
- This introduce unpredictable latency from a few microseconds to milliseconds.
  - Depends on from where classes are loaded
  - Static initialization
  - Number of classes loaded

A Java-conformant JVM must delay loading a class until it's first referenced by a program. Loading a class can take a variable amount of time depending on the speed of the medium (disk or other) the class is loaded from, the class's size, and the overhead incurred by the class loaders themselves. If tens or hundreds of classes need to be loaded, the loading time itself can cause a significant and possibly unexpected delay. Careful application design can be used to load all classes at application start-up, but this must be done manually because the Java language specification doesn't let the JVM perform this step early.

## JIT (Just In Time Compiler)

- Most modern JVMs initially interpret Java methods and, and later compile to native code.
- For a hard RT application, the inability to predict when the compilation will occur introduces too much nondeterminism to make it possible to plan the application's activities effectively.

When java was first introduced it was a interpreted language. But today nearly all JVMs use mixed mode execution. That means methods are initially interpreted but when the runtime has enough information about the code it is compiled to native code. HotSpot may even compile a method more than once to improve its performance when more information about the code is available.
Althoug this behavior maximizes throughput in a server application (where java is dominated today) introduce significant nondeterminism about performance of a method.
There are java compilers that produce native code directly for example gcj but moderm non-rt JVMs prefer JIT compilation because this makes lots of optimizations possible when compared to static compilation. For example polymorphic call to monomorphic call optimization, escape analyses.

## Garbage Collection

- Pros
  - Pointer safety,
  - leak avoidance,
  - Fast memory allocation:faster than malloc and comparable to alloca
  - Possible de-fragmentation
- Cons
  - Unpredictable pauses:Depends on size of the heap, number of **live** objects on the heap and garbage collection algorithm, number of cpus and their speed

Errors introduced by the need to manage memory explicitly in languages such as C and C++ are some of the most difficult problems to diagnose. Proving the absence of such errors when an application is deployed is also a fundamental challenge. One of the Java programming model's major strengths is that the JVM, not the application, performs memory management, which eliminates this burden for the application programmer.
On the other hand, traditional garbage collectors can introduce long delays at times that are virtually impossible for the application programmer to predict. Delays of several hundred milliseconds are not unusual

## Main Garbage Collection Features

- Stop-the-world or Concurrent
- Moving objects
- Generational
- Parallel

Concurrent means works concurrently with mutator threads

Parallel means more than one thread is used for collection

Moving :explain fragmentation

Generational:explain yound and old generation. Why this is necessary. Young generation collection and old generation collection. Fast/slow etc.

## Garbage Collection in HotSpot

- Serial Collector
  - -XX:+UseSerialGC
- Parallel-scavenging
  - -XX:+UseParallelGC
- Parallel compacting
  - -XX:+UseParallelOldGC
- Concurrent Mark and Sweep (CMS)
  - -XX:+UseConcMarkSweepGC

Different collectors can be used for different generation

Parallel-scavenging:like serial collector but uses multiple threads.Only works on young generation

Parallel compacting:like parallel-scavenging but works on yound and old generation

CMS works on only old generation. Parallel GC is used for young generation. It is not a compacting collector. To combat fragmentation assumes feature object size demands based on past allocations. Some malloc implementations also combats fragmentation using simillar techniques.

## Garbage First Collector (G1)

- Planned for JDK 7
- Low pause and high throughput soft real-time collector
- It is parallel and concurrent
- Performs compaction
- Devides heap to regions and further devides them to 512 bytes cards
- It is not a hard real-time collector

## Thread Management

- Although standard Java allows priority assignment to threads, does not require low priority threads to be scheduled before high priority ones
- Asynchronously interrupted exceptions
  - Similar to Thread.stop but this version is safe
- Priority Inversion may occur in standard Java/Operating systems
  - RTSJ requires Priority Inheritance

Standard Java provides no guarantees for thread scheduling or thread priorities. An application that must respond to events in a well-defined time has no way to ensure that another low-priority thread won't get scheduled in front of a high-priority thread.
Asynchronously Interrupted exceptions provides a safe way to stop a thread from another thread. But unlike Thread.stop this method stops the target thread in a controlled way.
Priority Inversion may occur in non-rt operating systems and standard java. RTSJ requires an operating system that implements Priority Inheritance but allows others like Priority Ceiling emulation.

## What is RTSJ?

- Designed to support both hard real-time and soft real-time applications
- Spec is submitted jointly by IBM and Sun
- Spec is first approved in 2002 (JSR 1)
- Minor updates started on 2005 (JSR 282)
- First requirements are developed by The National Institute of Standards and Technology (NIST) real-time Java requirements group

Java Real-Time Specification is shortly written as RTSJ. In 1998 a group of experts in real-time computer control was formed and coordinated by the National Institute of Standards and Technology (NIST) to draft requirements for real-time Java. The members were representatives from 50 different companies, government institutes and researchers from academic institutions. The main aim of the group was to develop a cross-disciplinary specification for real-time functionality that is expected to be needed by real-time applications written in Java programming language and being executed on various platforms. The group workshops at NIST produced nine core requirements for a Real-Time Java specification, together with number of derived sub-core requirements. This is the basis of the Real-Time Specification for Java (RTSJ). The weaknesses of the Java language specification for writing real-time applications are addressed in the core requirements.

# Expert Group of RTSJ

**Specification Lead**

Peter Dibble TimeSys Corporation

**Expert Group**

| | | |
|---|---|---|
| Ajile Systems | Apogee Software, Inc. | Belliardi, Rudy |
| Brosgol, Benjamin | Cyberonix | MITRE Corporation |
| Motorola | Nortel | NSI COM |
| QNX | Rockwell Collins | Sun Microsystems, Inc. |
| Thales Group | TimeSys Corporation | WindRiver Systems |

## RTSJ's Problem Domain

- If price of processor or memory is a small part of whole system RTSJ is a good choice
  - If processor and memory price is very significant when compared with the rest of the system RTSJ may not be a good choice because you will need slightly more powerful processor and larger memory
  - For very low footprint applications with very limited resources you may consider a non RTSJ compliant virtual machine like Aonix Perc Pico or you may even consider a hardware JVM.

If your application can tolerate some degree of non-determinism, then use a non-real-time virtual machine and tune it to obtain the response time you need, down to perhaps a 20 millisecond latency or so, but be aware that there will be occasional response time outliers. If you need response times below 20 milliseconds, even down to 30–70 microseconds, use Java RTS and its RTGC.

Achieving any degree of predictability requires trading off application throughput in various ways. Virtual machine selection and configuration therefore occurs along a predictability spectrum. As virtual machines evolve, we can expect to be able to specify a desired level of predictability and have the system configure itself automatically to achieve it. Until then, a degree of manual configuration will be necessary. Using Java RTS and RTGC minimizes the effort necessary to achieve sub-millisecond levels of predictability.

## Implementations

- IBM Webshere Real Time (RTSJ Compliant)
  - Works on real-time linux kernel
  - There is a soft real-time version
- SUN RTS (RTSJ Compliant)
  - Works on Solaris x86/Sparc or real-time linux kernel
- Aonix Perc (Not RTSJ Compliant but close)
  - Perc-Ultra works on nearly all platforms
  - They have a safety critical JVM (Perc-Raven)
- Apogee
  - Woks on nearly all platforms

# Main Features of RT Java Implementations

**Full Java SE compatibility and Java syntax**

A way to write programs that do not need garbage collection (New API)

High resolution timer (New API)

Thread priorities and locking (New API)

Asynchronous Event Handers (New API)

Direct memory access (New API)

Asynchronous Transfer of Control (New API)

AOT (A Head of Time) compilation (Not in RTSJ specification)

Garbage collection (Not in RTSJ specification)

## Real-Time Garbage Collectors

- Time based (Metronome of Websphere RT)
- Work based
- Henriksson’s GC (Garbage Collector of Sun’s Java RTS is based on this)

## Sun's RT Garbage Collector

- 3 modes of execution
- Non generational, concurrent and parallel collector
- In normal mode works with priority higher than non-rt threads but lower than any rt thread.
- When free memory goes below a certain threshold priority is increased to boosted priority but RT Threads (including the one with priority lower than GC) still continues to work concurently
- When free memory goes below a critical threshold.All threads whose priority is lower than GC boosted priority are suspended
- Most of its working can be tuned using command line switches like: NormalMinFreeBytes, RTGCNormalWorkers, RTGCBoostedPriority, BoostedMinFreeBytes etc.

It is a non-generational garbage collector. It operates on all heap at each execution. It has 3 modes of execution. It's has 3 modes of execution. In normal mode priority is lowest RT priority by default (can change this using –XX:RTGCNormalPriority). It works concurrently with other threads. If free memory goes below a certain threshold (can be configured with –XX:NormalMinFreeBytes), it switches to boosted mode. In this mode its priority increased to boosted priority (Can be changed with –XX:RTGCBoostedPriority). Although in this mode GC can preempt RT Threads with priority lower than GC boosted priority, it can still work concurrently with them. If free memory continues to drop to a critical threshold, GC switches to critical mode. This mode is called deterministic mode. In this mode its priority is still boosted mode priority but all non-rt threads and threads with priority lower than GC boosted priority are suspended. If a thread wich has priority higher than boosted GC priority makes memory allocation it can be served from a special region called critical reserved memory (size can be adjusted using –XX:RTGCCriticalBoundary). Number of GC threads in normal and boosted/critical mode can be adjusted using commandline switches.

# AOT, ITC and JIT

- Nearly all Java SE vendors use JIT (Just in time compiler) due to performance reasons
- Websphere RT uses AOT (Ahead of Time Compilation)
- SUN's Java RTS uses ITC (Initialization Time Compilation)

Refer following IBM Article for further information: Real-time Java, Part 2: Comparing compilation techniques

To use AOT or ITC compilation you have to provide list of classes and methods to be compiled ahead of time or at jvm startup. RTSJ implementations may provide tools to help you to generate this list.

# High-Resolution Timer

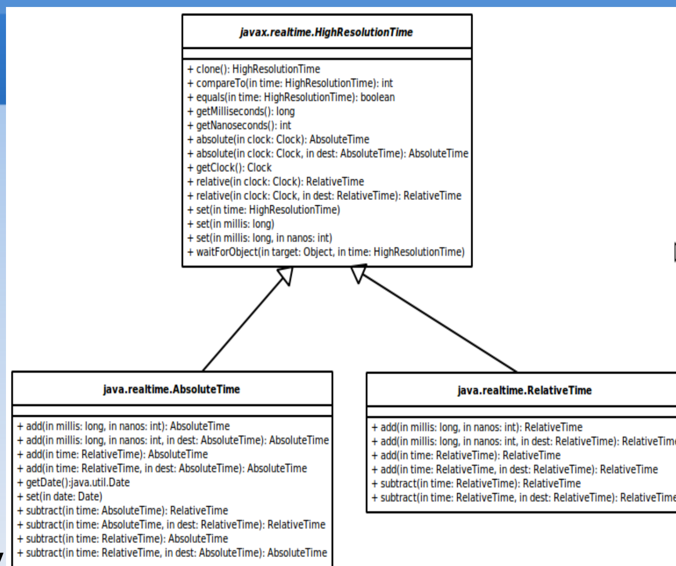
- A more powerful notion of time
  - AbsoluteTime
  - RelativeTime
- 84 bits value
  - 64 bits milliseconds and 32 bits nanoseconds
  - Range is 292 million years
- Every RTSJ implementation should have provide at least one high resolution clock accessible with *Clock.getRealTimeClock()*
- Resolution is system dependent
  - For example it is 200ns for SunFire v240 (Ref:Real-Time Java Programming by Greg Bollella)
  - Do not expect too much from a laptop

| **javax.realtime.Clock** |
|---|
| |
| + getEpochOffset(): RelatieTime<br>+ getRealtimeClock(): Clock<br>+ getResolution(): RelativeTime<br>+ getTime(): AbsoluteTime<br>+ getTime(dest:AbsoluteTime):AbsoluteTime() |

A High Resolution Timer is required for an RT System. RT-Posix requires one. If you look at getTime(AbsoluteTime) method you can see that this method enables you to get time without creating a new AbsoluteTime object. Reason for this will be clear when we learn other features of RTSJ.

# HighResolutionTime Base Class

- Base class for other 3 time classes.
- Define the interface and provides implementation of some methods
- This class and its subclass do not provide any synchronization




```
AbsoluteTime absolute(Clock clock)
AbsoluteTime absolute(Clock clock, AbsoluteTime dest)
int compareTo(HighResolutionTime time)
int compareTo(java.lang.Object object)
boolean equals(java.lang.Object object)
boolean equals(HighResolutionTime object)
long getMilliseconds()
int getNanoseconds()
int hashCode()
RelativeTime relative(Clock clock)
RelativeTime relative(Clock clock, RelativeTime time)
void set(HighResolutionTime time)
void set(long millis)
void set(long millis, int nanos)
static void waitForObject(java.lang.Object target,
        HighResolutionTime time) throws
        InterruptedException
```

# AbsoluteTime and RelativeTime

- AbsoluteTime represents a specific point in time
  - The string 03.12.2008 11:00:00.000 AM represents an absolute time
  - Relative to 00:00:00.000 01.01.1970 GMT
- RelativeTime represents a time interval
  - Usually used to specify a period for a schedulable object
  - Can be positive, negative or zero

## Priority Scheduler

- Normal scheduling algorithms try to be fair for task scheduling.
  - Even lower priority tasks are scheduled some times
- There are different real-time and non real-time schedulers
  - Earliest-deadline first, least slack time, latest release time etc.
- Fixed priority preemptive scheduler is most used real-time scheduler

Fixed priority preemptive scheduler does not change priority of the tasks automatically for this reason it is called fixed priority. Preemptive scheduler may preemt a task if a higher priority task is runnable.

## Thread Scheduling in RTSJ

- Does not specify a scheduling implementation
- Allow different schedulers to be plugged
- Required base scheduler should be a priority scheduler with at least 28 unique priorities
- Most implementations provide more than 28 priorities

More on this with Scheduling Parameters Slide

Note that RT-Posix required 32 unique priorities. These requirements are also required by RT-Posix

## What is Priority Inversion?

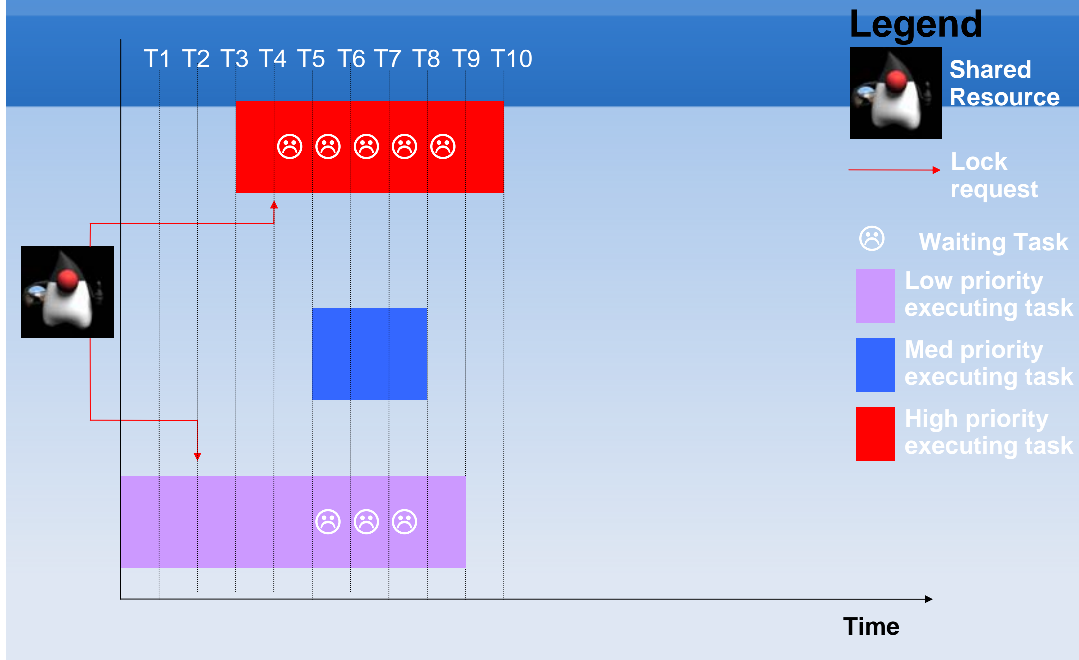
- Priority Inversion is the situation where a higher priority task waits for a low priority task.
- There are 2 generally accepted solution
  - Priority Inheritance (required by RTSJ)
  - Priority Ceiling Emulation (optional for RTSJ)

Explain priority inversion by drawing on the board.

The trouble experienced by the Mars lander "Mars Pathfinder"[1][2] is a classic example of problems caused by priority inversion in realtime systems. Software continously reseted by watchdog timer due to priority inversion in actual mission. Error occured once in test phase but can not be repeated and clasified as a hardware glitch.

Under the policy of priority inheritance, whenever a high priority task has to wait for some resource shared with an executing low priority task, the low priority task is assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task

With priority ceilings, the shared mutex process (that runs the operating system code) has a characteristic (high) priority of its own, which is assigned to the task locking the mutex. This works well, provided the other high priority task(s) that try to access the mutex does not have a priority higher than the ceiling priority.




T1:Low priority task is executing

T2:Low priority task acquire shared lock

T3:High priority task preempts low priority task

T4:High priority task requests the same shared resource
with low priority one and suspended. Low priority Task executes

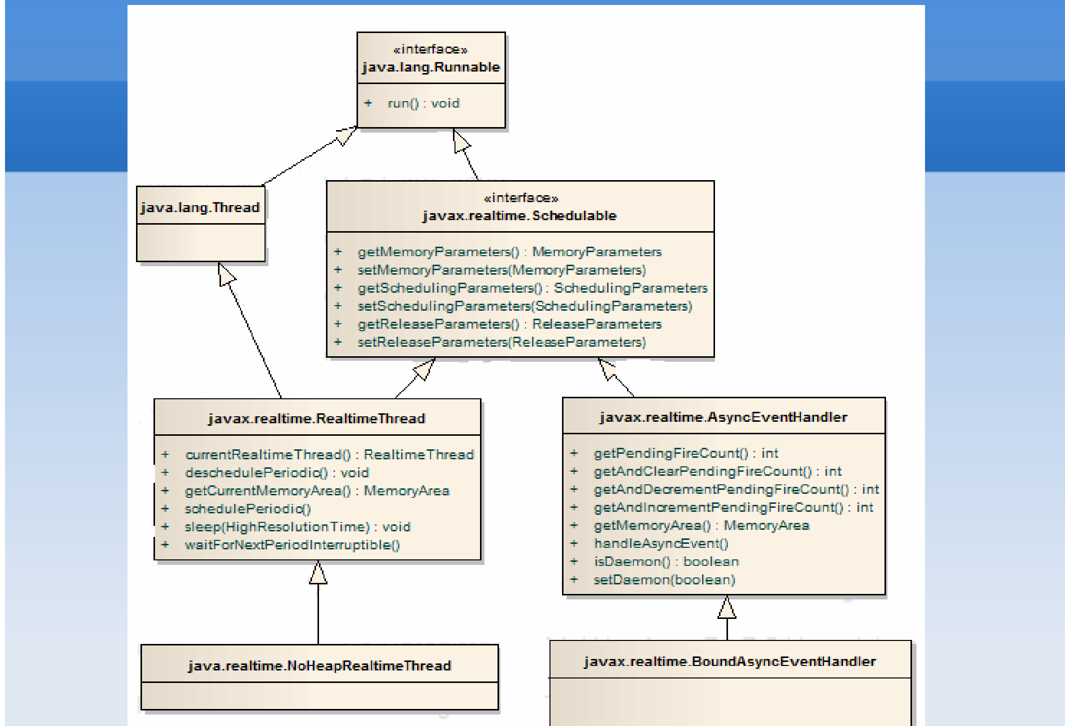
T5:Medium priority task preempts low priority task

T6:Medium priority task continues to execute

T7:Medium priority task executes and completes

T8:Low priority task executes and completes

T9:High priority task acuires the lock and completes




Real time threads are very similar to JLTs with some extra behaviors and constraints. They can do everything a JLT can do. You can specify several parameters parameters while creating them or at a later time.

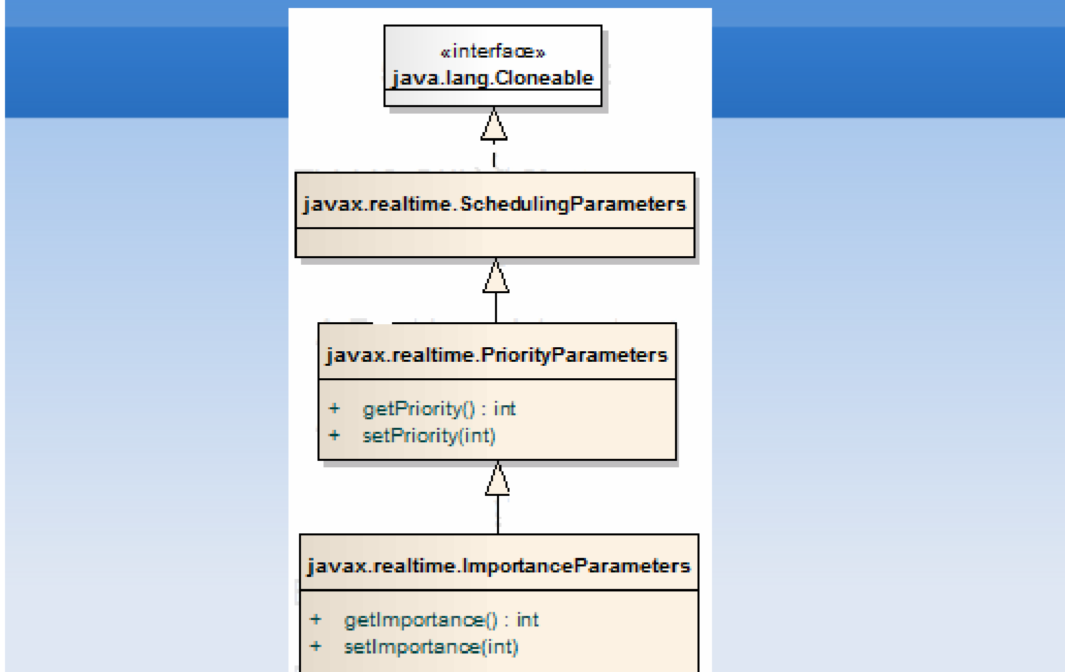
Most important parameters are as follows:

SchedulingParameters: This parameter is handled by underlying scheduler and tell scheduler how this Thread should be scheduled.

ReleaseParameters:As you will see at later slides, a thread can be periodic, aperiodic or sporadic.

MemoryArea: RTSJ defines other memory regios than heap. A real-time thread can perform allocation in these new memory regions. The memory region passed with this parameter will be default allocation context of this thread.

All parameters has a default value and null can be passed.




RTSJ defines a plugable scheduler architecture. Different schedulers may be plugged and each different scheduler may need different information as its scheduling parameters. RTSJ's default priority scheduler defines scheduling parameter as priority so a PriorityParameters class is included to use with priority scheduler.

Sometimes you may need to define an order between the task with the same priorities. Using ImportanceParameters you can define relative importance of tasks with the same priority. Base PriorityScheduler does NOT considers ImportanceParameters.

Number of available priorities depends on target operating system. You can expect minimum of 28 different priorities. On solaris there are 60, on linux there are 49.

## Different Task Types

- Periodic
- Aperiodic Tasks
- Sporadic

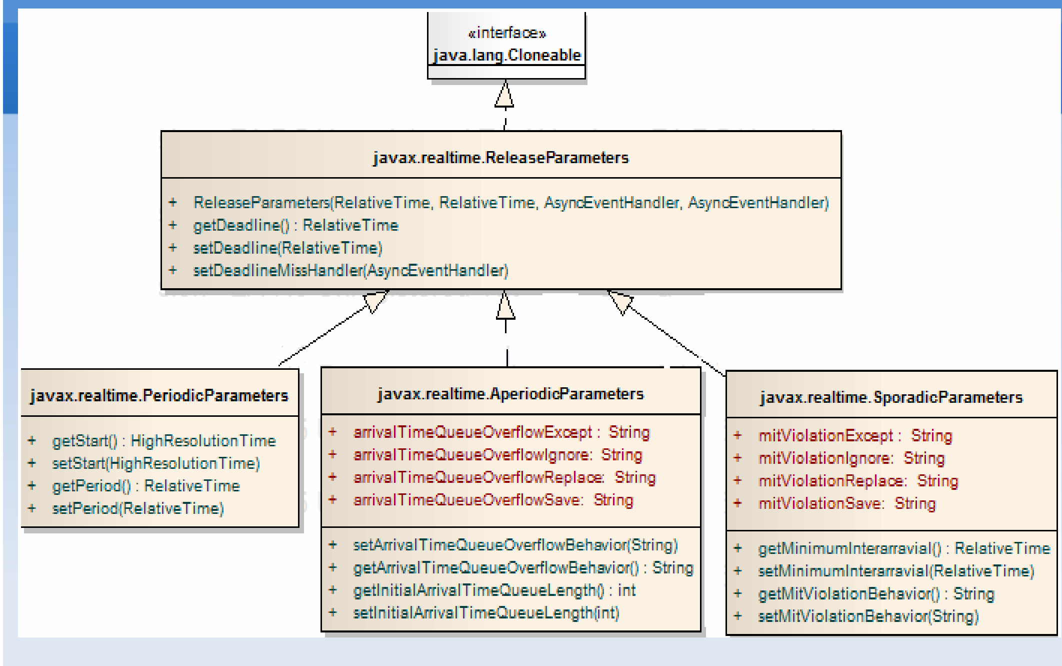
Periodic tasks occursat a known rate. It is easy to analyse feasibilty of meeting deadlines if you have all peridic task with known costs

Aperiodic tasks does not have any occurance rate. They may occur at any time.

Sporadic tasks are like aperiodic tasks but they have a known minimum inter occurance time.

Aperidic taks can be scheduled using a sporadic server task. Which executes at a scheduled rate and executes a periodic taks.

# Release Parameters




Release parameters determine whether a thread is periodic, aperiodic or sporadic. You can specify time with nanoseconds resolution. Actual resolution will be platform dependent and you should test on your target environment. For a periodic task there is a minimum period that can be specified. This value is plaform dependent and should be tested. Your target environment may allow you to change this value to a lower one.

All aperiodic events are put to a queue. Size of this queue can be specified using setInitialArrivalTimeQueueLength method. You can also specify what to do if this queue fulls. Valid optionas are throwing an exception, disgarding event, increasing queue size accordingly or replacing the last one on the queue.

Sporadic tasks are like aperiodic tasks. Instead this time you specify what to do task which does not obey their minimum inter arrival time using setMitViolationBehavior. Your valid options are the same.

# Sample Periodic Task

```
public class HelloWorld extends RealtimeThread {
    public static long[] times = (long[]) ImmortalMemory.instance().newArray(long.class, 100);

    public HelloWorld(PeriodicParameters pp) {
        super(null, pp);
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            times[i] = System.currentTimeMillis();
            waitForNextPeriod(); //wait for next period
        }
    }

    public static void main(String[] argv) {
        //schedule real time thread at every 100 milisecond
        PeriodicParameters pp = new PeriodicParameters(new RelativeTime(100, 0));
        HelloWorld rtt = new HelloWorld(pp);
        rtt.start();

        //wait real time thread to terminate
        try {
            rtt.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }

        //print results
        for (int i = 0; i < 100; i++) {
            System.out.println(times[i]);
        }
    }
}
```

## Periodic Execution vs Thread.sleep

```
While (true) {
  performPeriodicTask()
  Thread.sleep(period)
}
```

WRONG

**Real-time systems do not use a loop with a sleep in it to drive periodic executions**

A loop that uses sleep to execute once per period will cycle too slowly since every iteration of the loop uses sleep time plus the time to execute the code in the loop. Even measuring the time of the code in loop may not help. It is nearly impossible to get correct.

# Deadline Miss!

- For some applications a deadline should not be missed for any reason
- For most applications dealine miss is bad, but it is not a total failure and a recovery is possible
- In RTSJ there are 2 ways to detect a deadline miss
  - waitForNextPeriod() returns false **immediately**
  - Use a deadline miss handler which is an instance of AEH

If a thread misses its deadline due to system overload executing an extra event handler may overload system more and cause extra deadline misses. You should design your miss handlers as light-weight as possible. Note that execution of deadline miss handler is not synchronous. Miss handler may block the thread, or may work parallel to thread depending on the priority of the handler and number of CPUs.A thread that misses a deadline continues to execute until it calls waitForNextPeriod. If thread does not have a deadline miss handler, waitForNextPeriod returns immediately with false value to signal a deadline miss. An RT Thread wihch misses a deadline is automatically descheduled, its deadline miss handler may reschedule it, terminate it using interruption.

## NoHeapRealTime Thread

- Can not access heap
  - Can preempt GC immediately
  - If this rule is violated, MemoryAccessError is thrown
- Can use ScopedMemory or ImmortalMemory
- Should be created and started in a scoped or immortal memory
  - They can not be created from normal Threads

Since NHRT are not allowed to access heap, they can preempt garbage collector at any time without any additional latency. Although normal RT Threads can also preempt GC they should always wait GC to be at a safe preemption point.

They can not be created from normal threads because normal threads can not execute in scoped or immortal memory.

They can only use ImmortalMemory or ScopedMemory. If a NHRT tries to access heap, MemoryAccessError is thrown. Enforcing this rule at runtime introduce additional runtime overhead. But some of it can be avoided by JVM by analysing application at runtime.

## WaitFreeWriteQueue

- Intendent for exchanging data between real-time and non real-time part
- Real-time producer does not block when queueing data
- Multiple non-real time consumers may dequeue data

| javax.realtime.WaitFreeWriteQueue |
|---|
| + write(Object) : boolean |
| + force(Object) : boolean |
| + read() : Object |
| + size() : int |
| + isEmpty() : boolean |
| + clear() |
| + isFull() : boolean |
| + WaitFreeWriteQueue(int, MemoryArea) |

Sometimes hard real-time and non-realtime parts of the application need to change data. The important thing is not blocking a real-time thread due to a non real-time thread. To faciliate this wait free queues are provided.

The write method appends a new element onto the queue. It is not synchronized, and does not block when the queue is full (it returns false instead). Multiple writer threads or schedulable objects are permitted, but if two or more threads intend to write to the same WaitFreeWriteQueue they will need to arrange explicit synchronization. If write is successfull write method returns true. Otherwise it returns null. Force method is similar to write method. The difference is if queue is full, it overrides the lates element in the queue. Return value of force method show whether an element is overriden.

The read method removes the oldest element from the queue. It is synchronized, and will block when the queue is empty. It may be called by more than one reader, in which case the different callers will read different elements from the queue.

# WaitFreeReadQueue

- Intendent for exchanging data between real-time and non real-time part
- Real-time consumer does not block when reading data
- Multiple non-real time producers may queue data

| javax.realtime.WaitReadQueue |
| --- |
| + write(Object) : boolean |
| + waitForData() |
| + read() : Object |
| + size() : int |
| + isEmpty() : boolean |
| + clear() |
| + isFull() : boolean |
| + WaitFreeReadQueue(int, MemoryArea) |

The write method appends a new element onto the queue. It is synchronized, and blocks when the queue is full. It may be called by more than one writer, in which case, the different callers will write to different elements of the queue.

The read method removes the oldest element from the queue. It is not synchronized and does not block; it will return null when the queue is empty.Multiple reader threads or schedulable objects are permitted, but if two or more intend to read from the same WaitFreeWriteQueue they will need to arrange explicit synchronization.

## Memory Regions

- Heap Memory
  - Same as in Java SE
  - Can be accessed with javax.realtime.HeapMemory
- Scoped Memory
  - Created and sized at development time
  - Can be accessed with javax.realtime.ScopedMemory
  - Can not be garbage collected; reference count to ScopedMemory object is used. Finalize method of objects are called
  - Can be stacked
- Immortal Memory
  - Can be accessed with javax.realtime.ImmortalMemory
  - Only one instance exist and size is determined at development time.
  - All static data and allocations performed from static initializers are allocated in Immortal memory. Interned Strings are also allocated in immortal memory
- Physical Memory
  - There are LTPhysicalMemory, VTPhysicalMemory, and ImmortalPhysicalMemory

Each schedulable has an allocation context. It can be one of the 4 defined main memory regions.

ImmortalMemory is a memory resource that is unexceptionally available to all schedulable objects and Java threads for use and allocation. An immortal object may not contain references to any form of scoped memory. Object in immortalMemory are never garbage collected. All static data goes into this region.

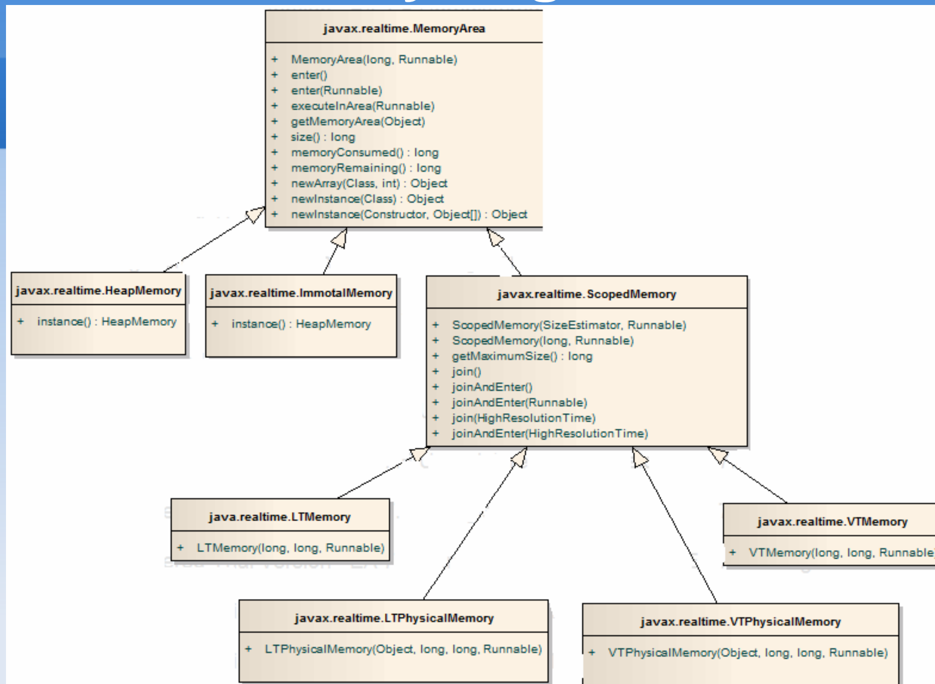
Objects allocated in scoped memory are freed when (and only when) no schedulable object has access to the objects in the scoped memory. When a ScopedMemory area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents (it's backing store) is allocated from memory that is not otherwise directly visible to Java code; e.g., it might be allocated with the C malloc function. This backing store behaves effectively as if it were allocated when the associated scoped memory object is constructed and freed at that scoped memory object's finalization. The enter() method of ScopedMemory is one mechanism used to make a memory area the current allocation context. The other mechanism for activating a memory area is making it the initial memory area for a real-time thread or async event handler.

An instance of ImmortalPhysicalMemory allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same restrictive set of assignment rules as ImmortalMemory memory areas, and may be used in any execution context where ImmortalMemory is appropriate.

LTMemory represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the memory area's *initial* size. Execution time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available. Objects in LTMemory can be safely accessed from NoHeapRealTimeThread.

VTMemory is similar to LTMemory except that the execution time of an allocation from a VTMemory area need not complete in linear time.

# Memory Regions




MemoryArea is the abstract base class of all classes dealing with the representations of allocatable memory areas, including the immortal memory area, physical memory and scoped memory areas.

# Raw Memory Access

- Models a range of physical memory as a fixed sequence of bytes
- Allows device drivers to be writen in java
- All raw memory access is treated as volatile, and *serialized*

| javax.realtime.RawMemoryAccess |
| --- |
| + RawMemoryAccess(Object, long, long) |
| + getByte(long) : byte |
| + getBytes(long, byte[], int, int) |
| + getInt(long) : int |
| + getInts(long, int[], int, int) |
| + getLong(long) : long |
| + getLongs(long, long[], int, int) |

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

The RawMemoryAccess class allows a real-time program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software. List of methods in the diagram are not complete.

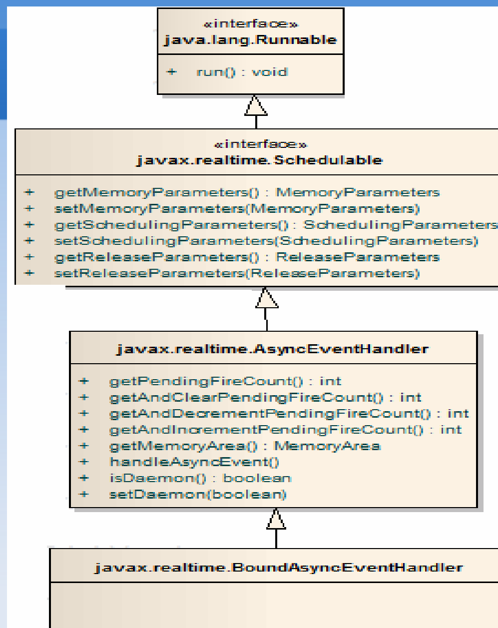
# Raw Memory Access Cont

```
private final long DEVICE_BASE_ADDRESS = xxxxxx;
private final long CTRLREG = 0;
private final long STATREG = 4;
………

public void init() {
    RawMemoryAccess device = new RawMemoryAccess(type, DEVICE_BASE_ADDRESS);
    device.setInt(CTRL_REG, MY_COMMAND); //send command to device
    while (device.getInt(STATREG) != 0); //wait for device to response
}
```

# Async Events

- Many of the processing in RT systems are triggered by internal or external events
- You don't need to manage threads
- Hundreds of AEHs may share a pool of threads
- BoundAsyncEventHandler has always bounded to a dedicated thread




- How binding is performed is not specified in RTSJ. A bind method is defined but possible values of its string argument are not defined and platform specific. You may create your own event objects by extending from AsyncEvent. They are not only for external events.
- AEHs are designed to execute a few lines of code, then exit. They are not expected to block, sleep, or change its scheduling parameters.

# Handling Posix Events

- Use POSIXSignalHandler
  - Ex:

```
.....

class SigintHander extends AsynchEventHandler {
    public SigintHandler() {
        //set it to highest priority
        setSchedulingParameters(new PriorityParameters(RTSJ_MAX_PRI);
    }
    public void handleAsynchEvent() {
        //handle user specified signal
    }
}

......
//add handler to posix predefined posix signal
POSIXSignalHandler.addHandler(PosixSignalHandler.SIGUSR1, sigintHandler)


Use kill -s SIGUSR1 PID to test
```

# Time Triggered Events

- **OneShotTimer :** execute handleAsyncEvent method once at the specified time
- **PeriodicTimer :** execute handleAsyncEvent method repeatedly at specified interval. A periodicTimer and a AEH combination is roughly equivalent to Periodic Threads.
- **Enable/Disable Timer :** A disabled timer is still kicking. But it does not generate events. When enabled again, it continues like never disabled.

## Javolution Library (http://javolution.org)

- High performance and time deterministic (util/lang/text/io/xml)
- Struct and Union base classes for direct interfacing with native applications
- NHRT Safe
- Pure Java and less than 300Kbytes
- BSD License

1. You don't need to guess the capacity of a TextBuilder, FastTable or a FastMap, their size expand gently without ever incurring expensive resize/copy or rehash operations (unlike StringBuilder, ArrayList or HashMap).

3. Javolution classes are fast, very fast (e.g. Text insertion/deletion in O[Log(n)] instead of O[n] for standard StringBuffer/StringBuilder).
4. All Javolution classes are hard real-time compliant and have highly deterministic behavior (in the microsecond range). Furthermore (unlike the standard library), Javolution is RTSJ safe (no memory clash or memory leak when used with Java Real-Time extension).
5. Javolution makes it easy for concurrent algorithms to take advantage of multi-processors systems.
6. Javolution's real-time collection classes (map, list, table and set) can be used in place of most standard collection classes and provide numerous additional capabilities.
7. Any Java class can be serialized/deserialized in XML format in any form you may want, also no need to implement Serializable or for the platform to support serialization
8. Javolution provides Struct and Union classes for direct interoperability with C/C++ applications.
9. Javolution runs on any platform from the simplest J2ME CLDC 1.0 with no garbage collector to the latest J2EE 5.0 with parameterized types.
10. Javolution is a pure Java Solution (no native code), small (less than 300 KBytes jar file) and free; permission to use, copy, modify, and distribute this software is freely granted, provided that copyright notices are preserved (BSD License).

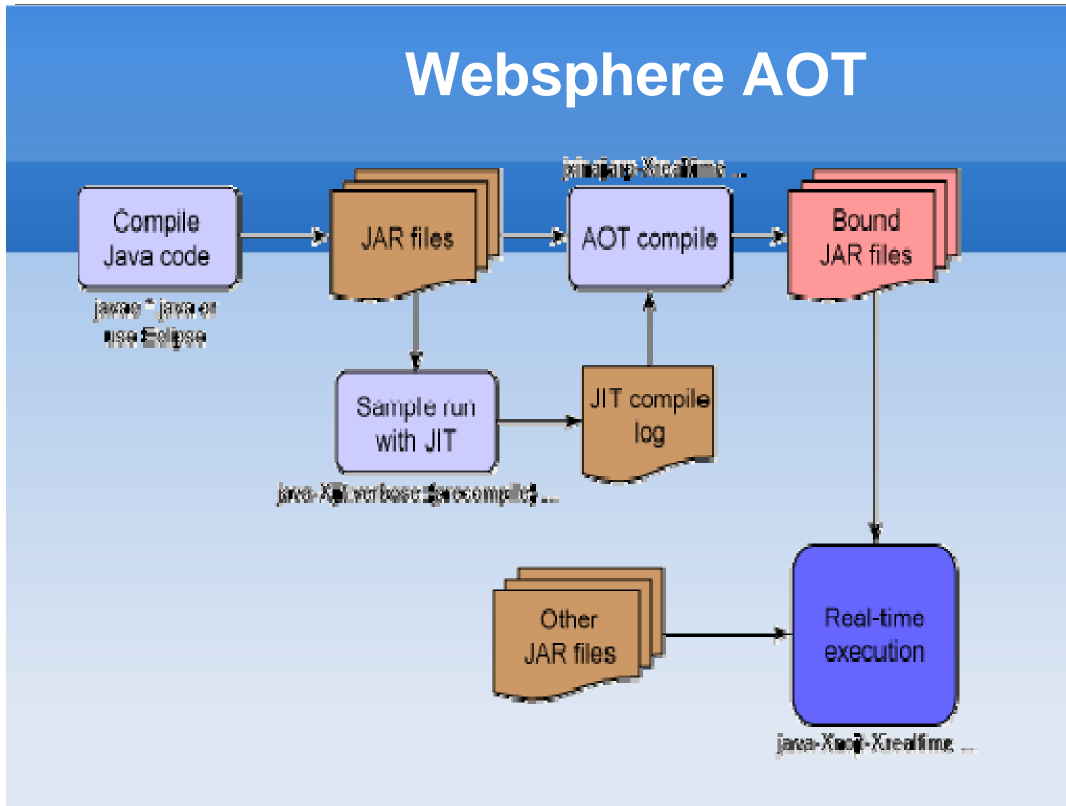
# IBM WebSphere Real Time

- Runs on linux with real time patches applied to linux kernel on x86 platforms
- Has AOT and JIT Compilers
- Shared classes support
- Contains Metronome: a real-time garbage collector
- Full RTSJ 1.0.2 and Java SE 6.0 support
- A well defined list of NHRT safe classes

## Metronome Garbage Collector

- Uses a time based method for scheduling
- Applications threads are given a minimum percentage of time (utilization)
  - User supplied at startup
- Uses a new two-level object model for arrays called arraylets

The reason for scheduling against time instead of allocation rate is that allocation is often uneven during an application's execution.
Metronome divides time into a series of discrete quanta, approximately 500 microseconds but no more than 1 millisecond in length, that are devoted to either GC work or application work.
Arraylets break up large arrays into smaller pieces to make large array allocations easier to satisfy without defragmenting the heap. The arraylet object's first level, known as the spine, contains a list of pointers to the array's smaller pieces, known as leaves. Each leaf is the same size, which simplifies the calculation to find any particular element of the array and also makes it easier for the collector to find a suitable free space to allocate each leaf. Breaking arrays up into smaller noncontiguous pieces lets arrays be allocated within the many smaller free areas that typically occur on a heap, without needing to compact.



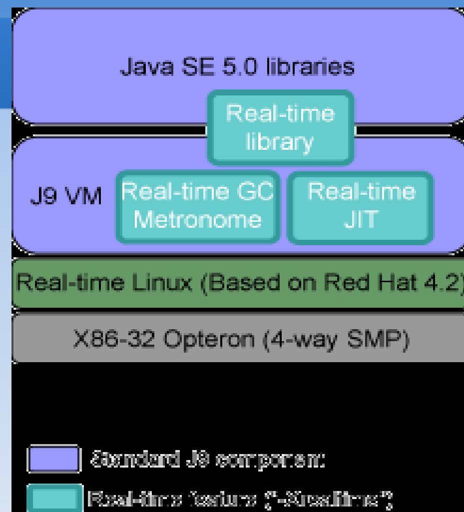

All the classes in a jar file can be compiled or only hot methods can be compiled using information obtained from a previous run of the application.
Although AOT code enables more-deterministic performance, it also has some disadvantages. The JXEs used to store AOT code are generally much larger than the JAR files that hold the class files because native code is generally less dense than the bytecodes stored in class files.A second disadvantage is that AOT-compiled code, though faster than interpreted code, can be substantially slower than JIT-compiled code. To avoid nondeterministic performance effects, neither the JIT compiler nor the AOT compiler provided in WebSphere Real Time applies the aggressively speculative optimizations generally applied by modern JIT compilers.

## SUN RTS

- Achieves maximum latencies of 15 microseconds, with around 5 microseconds of jitter.
- Runs on real-time linux and Solaris 10
- Has a real-time garbage collector

# IBM WebSphere Real Time

- Runs on linux with real time patches applied to linux kernel
- Has AOT and JIT Compilers
- Contains Metronome real-time garbage collector
- RTSJ and Java SE 5.0 Compliant



# Real World Projects:J-UCAS X-45C

Figure 1

The mission planning software for the J-UCAS X-45C unmanned aircraft was written in the Java language and deployed on a real-time virtual machine. The code was developed as a collaboration between engineers at Boeing and British Aerospace Engineering (BAE).

# Real World Projects:FELIN

Figure 2

The FELIN project is a helmet-mounted personal digital assistant to help infantry communicate with each other and with commanders, and to provide navigation and situational awareness aids. This application software, developed in France by Sagem, was written in Java and deployed on a real-time virtual machine.

# Real World Projects:DDG-1000

Figure 1

For the DD(X) destroyer program—now called DDG-1000—software developers have pushed the envelope of Java real-time garbage collection technology. Release 4 of the DD(X) software environment will switch completely to a real-time Java VM.

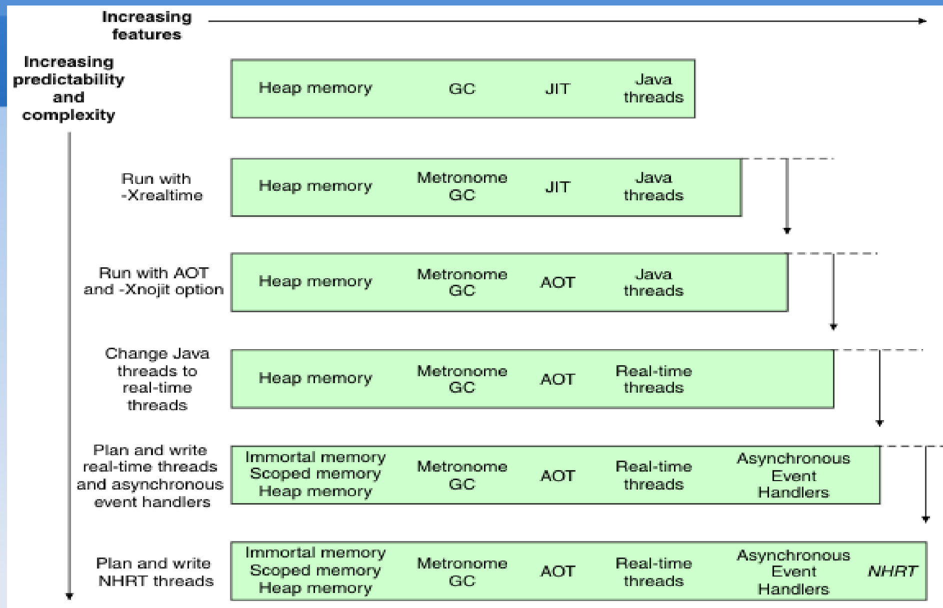
# Real World Projects:ScanEagle

This milestone marked the first flight using the RTSJ on an UAV and received the Java 2005 Duke's Choice Award for innovation in Java technology.

## Is RTSJ Required for You?

- If your application can tolerate some degree of indeterminism use standard JVM and tune it to milliseconds level
- Only if you fail the first approach use RTSJ.
  - Try to meet your timing constrains with real-time garbage collector (if available) without using advance/complex features like scoped memory
  - If first approach fails make use of NonHeapRealTimeThread, ImmortalMemory, ScopedMemory

# A comparison of the features of RTSJ with the increased predictability




**From IBM WebSphere Real Time Manual**

## Safety Critical Java (JSR 302)

- A subset of RTSJ that can be certified DO-178B and ED-128B.
- Most probably garbage collector will not be available (not needed)
- Will be targeted to Java ME platform, because Java SE and Java EE are too complex to be certified.

Java byte code will be converted to target platform prior to deployment and certification. Most probably garbage collector will not be available because it is very complex to certify and not needed by this kind of applications where resources are statically allocated.
A set of tools for analysing source code for verification will be provided

# Expert Group of JSR 302

- **Specification Lead:**C. Douglass Locke (POSIX 1003.4 Real-Time Extensions Working Group)
- Expert Group

| | | |
|---|---|---|
| Aicas GmbH | Aonix North America, Inc | Apogee Software, Inc. |
| AXE, Inc | Boeing | DDC-I, Inc |
| IBM | Siemens AG | Rockwell Collins, Inc |
| | Sun Microsystems | The Open Group |
| | | |

Aonix safety critical JVM is based on JSR 302. Selected to be evaluated in DIANA project
that aims to evaluate new tools to modernize tool chains used in avionics.
Aonix safety critical JVM (Raven) will be also evaluated in Taranis,
a technology demonstration program for enhanced aerial vehicles for the
UK Ministry of De fence

## Resources

Real-Time Java Platform Programming by Peter C. Dibble

Real-Time Java Programming with Java RTS by Greg Bollea

RTSJ Main Site (www.rtsj.org)

IBM's Developerworks Articles (http://www.ibm.com/developerworks/)

SUN RTS (http://java.sun.com/javase/technologies/realtime/reference.jsp)

Deniz Oğuz's blog (www.denizoguz.com)

## Resources Cont. (JavaOne Presentations in 2008)

- TS-4797 Fully Time-Deterministic Java Technology
  - Explains Javalution Library
- TS-5767 Real-Time Specification for Java (JSR 1)
  - Explains status of RTSJ. Join presentation from SUN, IBM, Locke Consulting (JSR 302 spec lead)
- TS-5925 A City-Driving Robotic Car Named Tommy Jr.
  - An autonomous ground vehicle fully powered by Java.
- TS-5609 Real Time: Understanding the Trade-offs Between Determinism and Throughput