# IN325 Real-Time Programming Languages

Real-Time Specification for Java

Christophe Garion

DMIA – ISAE

# License CC BY-NC-SA 3.0

# Acknowledgment

Some slides are borrowed from Éric Noulard's lecture on Real-Time Programming with Java.

Most of the examples from the RTSJ part are borrowed from Dibble 2008.

Dibble, Peter C. (2008).
**Real-Time Java Platform Programming**.
2nd edition.
BookSurge Publishing.
http://www.rtsj.org/RTJPP/rtjpp.html.

# Materials and lab sessions

All materials for this lecture including:

- slides
- lab sessions
- bibliography

are available on `http://www.tofgarion.net/lectures/IN325`

For lab sessions, you will use you own SVN repository for the lecture: `https://eduforge.isae.fr/repos/IN325/your.login/RTSJ`

# Bibliography I

Gosling, J. et al. (2013).
**The Java Language Specification**.
Java SE 7 Edition.
Java Series.
Addison-Wesley Professional.
`http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`.

Lindholm, T. et al. (2013).
**The Java Virtual Machine Specification**.
Java SE 7 Edition.
Java Series.
Addison-Wesley Professional.
`http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`.

# Bibliography II

Dibble, Peter C. (2008).
**Real-Time Java Platform Programming**.
2nd edition.
BookSurge Publishing.
http://www.rtsj.org/RTJPP/rtjpp.html.

Goetz, B. (2006).
**Java concurrency in practice**.
Addison-Wesley.
http://jcip.net/.

# Quote. . .

> *If Java had true garbage collection, most programs would delete themselves upon execution.*

Robert Sewell

# Outline

1 - Concurrency in Java

2 - RTSJ: Real-Time Specification for Java

# 1 - Concurrency in Java

1. **Threads in Java**
2. **Java concurrency API**

# Why concurrency? We are speaking about RT!

## Do not mix up!

Real-time is not about being fast but being **on time**.

➥ **on time** for some external **concurrent** event...

- meet someone else, a **rendez-vous**
- do not miss the train, an **absolute deadline**,
- the mason should finish the house before the painter come, a dependency **constraint**
- the ABS computer of the car should control the brakes before sliding, a **time latency**.

# Object orientation and concurrency

Object-Orientation (mainly classes, objects and methods) brings questions on how concurrency should be integrated with OO concepts:

- should one process/thread be mapped to exactly one object?
- should a thread be an object?
- is it possible to call object method concurrently?
- should the concurrency concept be builtin the language?
  - C answer is no, use a thread library (POSIX Thread may be),
  - C++ initial answer is bo, but latest C++ standard in 2011 changed that with the addition of std::thread standard library.
  - Java answer is yes, we will see.
  - Go language (`http://golang.org/`) answer is yes: goroutine and channels.
- Could the concurrency construct be orthogonal to the language?
  - ➥ yes, see OpenMP (`http://openmp.org`), give high level concurrency indications as comments (C, C++, Fortran)

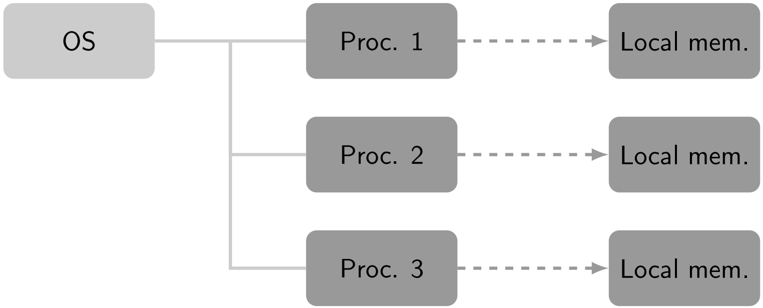
# Outline of part 1 - Concurrency in Java

1. **Threads in Java**
   - Threading API in Java: basics
   - Communication between threads
2. **Java concurrency API**

# Introduction: processes

A process is a set of instructions to execute, a memory space and eventually other resources (sockets, files, . . . ).

Operating systems allows to execute "simultaneously" several processes and to schedule them.

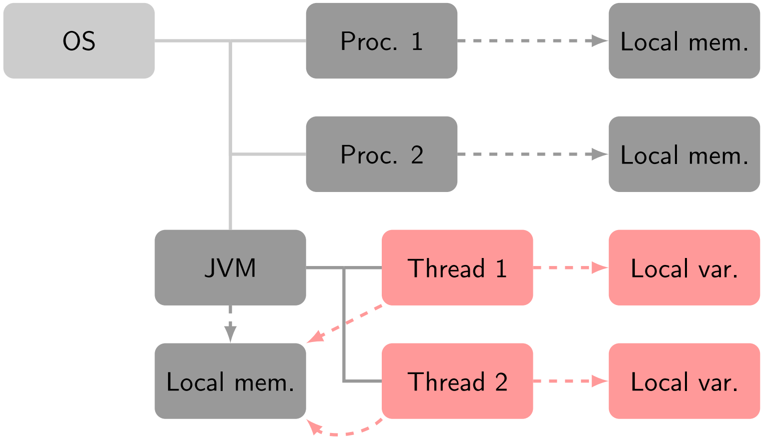


Processes does not normally share memory. They have their own stack and address space.

# Introduction: threads

**Thread** is an abbreviation of *thread of control*. We can also speak about lightweight process.

A thread is executed within a process. It shares memory space with the other threads of the process.

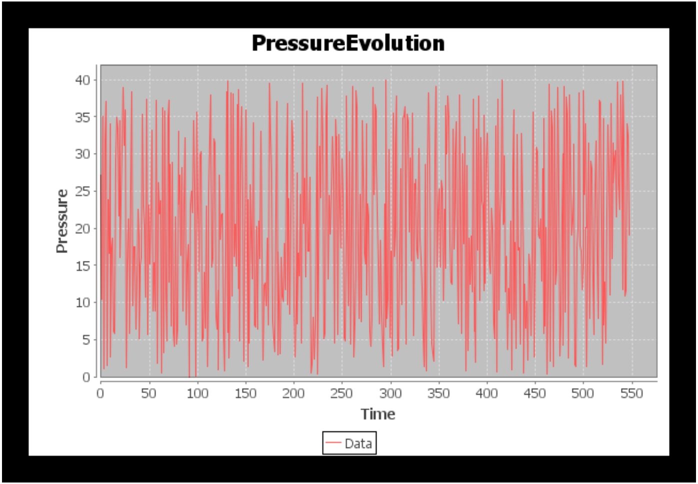


# Outline of part 1 - Concurrency in Java

1. **Threads in Java**
   - Threading API in Java: basics
   - Communication between threads
2. **Java concurrency API**

# A small example. . .

Graphical component that displays datas in RT:



# A small example: which threads?...

Application threads:

**DataWrapper**

- gets the data
- executed every 10 ms

**DVRefresher**

- refresh the GUI
- executed every 100 ms

But also...

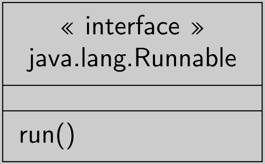
- Swing event thread
- threads created by `invokeLater`

**N.B.**

- `javax.swing.Timer` should have been used
- a MVC with listeners should have been more judicious...

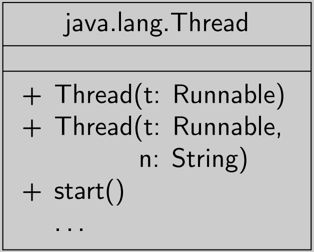
# Separating control from application

## Application


« interface »
java.lang.Runnable

run()

## Control


java.lang.Thread

+ Thread(t: Runnable)
+ Thread(t: Runnable,
  n: String)
+ start()
…

# The Runnable interface

The Runnable interface represents objects whose behavious should be executed by an active thread.

This interface represents the **fonctional** part of the thread.

## API of **Runnable**

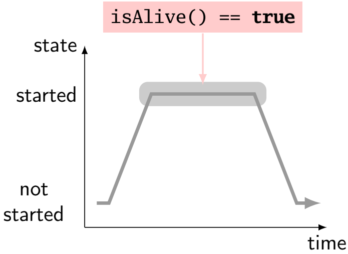
- **public void** run()

# The Thread class

The Thread class represents the thread **control flow**. It implements the Runnable interface.

## (Partial) API of **Thread**

- **void** start()
- **boolean** isAlive()
- **void** join()



## N.B.

You can also extends the Thread class to create an application, but. . .

# On our example

**DataFeeder.java**

```java
package fr.supaero.dv;

public class DataFeeder implements Runnable {

  private DataWrapper dw;
  private long delta;
  private int fakeTime;

  public DataFeeder(DataWrapper dw_, long delta_) {
    this.dw = dw_;
    this.delta = delta_;
  }

  @Override public void run() {
    while (true) {
      this.dw.add(this.fakeTime++, Math.random() * 40);

      try {
        Thread.sleep(this.delta);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

# On our example

**DVRefresher.java**

```java
public class DVRefresher implements Runnable {

  private DataVisualization dv;
  private long delta;

  public DVRefresher(DataVisualization dv_, long delta_) {
    this.dv = dv_;
    this.delta = delta_;
  }

  @Override public void run() {
    while (true) {
      SwingUtilities.invokeLater(new Runnable() {
          @Override public void run() {
            dv.refresh();
          }
        });

      try {
        Thread.sleep(this.delta);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

# Other methods in `Thread` API

## Thread interruption

- `static void sleep(long time)`
- `static void yield()`

Beware, those methods are not synchronized. For instance

```
while (!condition) {
  Thread.sleep(1000);
}
```

could never end!

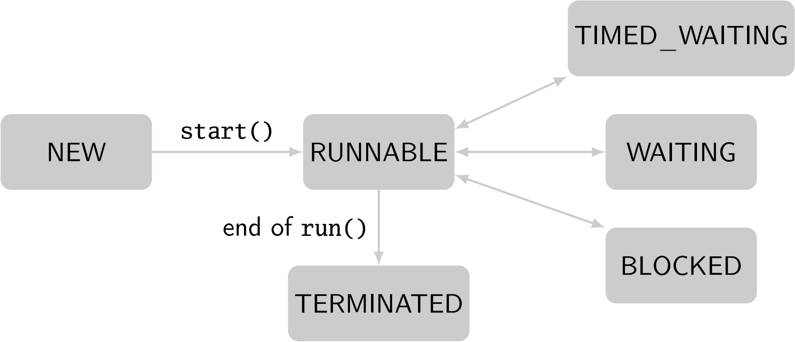
## Why?

The compiler does not have to refresh the cache or reload values (i.e. it can read `condition` only one time!), cf. Gosling et al. 2013.

# Other methods in Thread API

## Thread state

- `Thread.State getState()`



# Other methods in `Thread` API

## Threads groups

- `ThreadGroup getThreadGroup()`
- `int getPriority()`
- `void setPriority(int priority)` using `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`

## Other methods...

- `void setName(String name)`
- `String getName()`
- `Thread currentThread()`
- `void interrupt()`
- `boolean isInterrupted()`
- `void setDaemon(boolean on)`
- `boolean isDaemon()`

# Matrices multiplication ✎

### Exercise 1

Use thread to accelerate matrices multiplication (OK, not very original ☺).

# Outline of part 1 - Concurrency in Java

1. **Threads in Java**
   - Threading API in Java: basics
   - Communication between threads
2. **Java concurrency API**

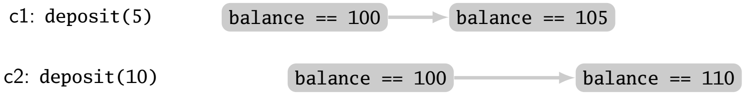
# Communication between threads

## Information sharing

The Java threads share the same address space inside the JVM and thus can share objects references and access directly classes features.

## Definition (Critical section)

A **critical section** is a code portion such that it cannot be executed simultaneously by two threads.



c1: deposit(5)

balance == 100 → balance == 105

c2: deposit(10)

balance == 100 → balance == 110

# Which operations are safe in Java?

## Principle (atomicity)

Affectation of variables of primitive types other than **`double`** or **`long`** is atomic.

On the other hand, **threads can share values and have a local copy of those values** (cf. Gosling et al. 2013).

You do not have any guarantee on the fact that a variable value read from a thread has taken into account the changes made by another one...

## Syntax (volatile)

```
visibility volatile type field;
```

The **`volatile`** keyword can be used to specify that a field should be "synchronized" at every modification.

# Hoare monitor

## Definition (Hoare monitor)

A Hoare monitor is an object that can be used **safely** by several threads.

Hoare, C. A. R. (1974).
“Monitors: an operating system structuring concept”.
In: **Communications of the ACM** 17.10,
Pp. 549–557.

# Hoare monitor

## Monitors in Java

Java associates to **every** `Object` instance a monitor.

`Object` has several methods associated to this monitor:

- **void** `wait()`: the current thread waits until another thread calls the `notify` or `notifyAll` methods on this object (+ temporized versions)
- **void** `notify()`: awakes a thread waiting on the object monitor
- **void** `notifyAll()`: awakes the threads waiting on the object monitor

# Hoare monitor

## Syntax (**synchronized**)

**synchronized** is a Java keyword that can be used:

- in a method declaration

```
public synchronized void deposit(double money)
```

- in a code block for a particular `Object` instance

```
synchronized(instance) {
  //code
}
```

# Hoare monitor

## Definition (**synchronized** semantics)

- only one thread can acquired the lock on an object monitor
- when a thread wants to execute a synchronized method or code block, it must first acquire the lock on the corresponding object monitor
- a thread that cannot acquire a lock awaits for this lock
- when finishing the execution of a synchronized method or code block, the executing thread release the lock on the corresponding object monitor

# The bank account ✎

## Exercise 2

Create a synchronized bank account (again, not very original ☺).

# The `wait`, `notify` and `notifyAll` methods

## `wait`, `notify` and `notifyAll`

- **void** `wait()`: **wait** for a condition. This method **must** be used inside a synchronized block or method.
  A *thread* calling `wait` must acquire the lock on the corresponding object monitor (beware of synchronized blocs!). The thread goes then in `WAITING` state and releases the lock.
- **void** `notify()`: **notifies a thread awaiting** for a condition. This method **must** be used inside a synchronized block or method. The thread must have the lock on the corresponding object monitor.
  **You cannot choose the notified thread (cf. JLS)!**
- **void** `notifyAll()`: **notifies all threads** awaiting for a condition. This method **must** be used inside a synchronized block or method. The thread must have the lock on the corresponding object monitor.

# The `wait`, `notify` and `notifyAll` methods

## Precisions on `wait`

- when entering in the `wait()` method, the lock on the object monitor is released
- the lock is **acquired just before the end** of the `wait` method
- the `wait` method is overloaded: `void wait(long timeout)`. This method returns after a length of `timeout` milliseconds, even if no notification has been produced
- the main difference with the method `sleep` is that the lock is released in the case of the `wait` method
- you should **always** put the `wait` method in an **infinite** loop testing the notification condition

# Producers/consumers

### Exercise 3

Create a producer/consumer framework with extra requirements.

All complaints for the extra requirements should be addressed to J. Hugues ☺

# Deprecated methods of `Thread` API

## Deprecated methods

- **void** `destroy()`: never implemented
- **void** `suspend()`
- **void** `resume()`
- **void** `stop()`

Oracle (2013).
**Java Thread Primitive Deprecation**.
http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html.

# Outline of part 1 - Concurrency in Java

1. Threads in Java

2. **Java concurrency API**

# Concurrency API to Java 1.4

## Problems for concurrency

- collections (except `Vector` and `Hashtable`) are not synchronized
- fail-fast mechanism for iterators on collections
- instances of *wrapper* classes (`Integer`, `Double` etc.) cannot be updated atomatically
- only lock notion: monitors

Beware particularly on collections that are not synchronized.

## How to create synchronized collections

Use static methods from `java.util.Collections`:

- `<T> Collection<T> synchronizedCollection(Collection<T> c)`
- `<T> List<T> synchronizedList(List<T> l)`
- ...

# Exercise on collections ✎

## Exercise 4

Use various implementations of collections with threads.

# New concurrency API from Java 5

## java.util.concurrent

- **synchronized collections**
- **executors**: allow to create subsystems with the same characteristics than threads
- **synchronizers**: semaphores, barriers etc.
- **timing** with nanosecond precision
- `java.util.concurrent.atomic`: types that can be updated atomically (`AtomicBoolean`, `AtomicInteger`, etc.)
- `java.util.concurrent.locks`

Grazi, V. (May 3, 2012).
**Java Concurrent Animated**.
`http://sourceforge.net/projects/javaconcurrenta/`.

# Concurrent collections

### `ConcurrentHashMap`

- no lock retained during retrieval operations like in synchronized collections
- lock striping
- iterators are not *fail-safe*, but weakly coherent, they do not throw `ConcurrentModificationException`
- atomic operations: `putIfAbsent`, `remove`, `replace`

# Concurrent collections

## CopyOnWriteArrayList

- operations modifying the list use a **copy** of the list
- when iterating on a list, the elements returned are those present in the list at the iterator's creation
- no fail-safe behaviour of iterators
- to be used with list that are not often modified

## CopyOnWriteArraySet

- *idem* but for a synchronized Set

# Concurrent collections

## BlockingQueue

| Operation | Exception | Special value | Blocks | Time Out |
|---|---|---|---|---|
| **Insert** | add | offer | put | offer |
| **Remove** | remove | poll | take | poll |
| **Examine** | element | peek | | |

- LinkedBlockingQueue, ArrayBlockingQueue: FIFO
- PriorityBlockingQueue: with priority using Comparable
- SynchronousQueue: handoff, RV mechanism in Ada

## BlockingDeque

- double ended queue

# Synchronizers

## Principle

Coordinate the control flow of threads using the state of the synchronizer.

A thread arriving on a synchronizer can pass or wait given the synchronizer's state.

# Synchronizers

## Latches

All threads arriving on a latch must wait that the latch reaches to its final state.
When the final state is reached, all threads are “freed”.
When the final state is reached, the state of the latch cannot be changed.

## CountDownLatch

- use a counter
- **void** await
- **void** countDown()

# Synchronizers

## FutureTask<E>

- represents a result of type E that will be computed in the future
- implements Runnable
- the computation is represented by a Callable<E> instance which have a method call (a Runnable instance can also be used)
- the threads calling the get method are blocked until call returns
- the result is returned to all blocked threads

# Synchronizers

## Semaphore

- represents an available number of resources
- **void** acquire()
- **void** release()
- + some variants

Can be used for instance to manage a pool of resources.

# Synchronizers

## CyclicBarrier

- blocks threads until a certain number of threads have achieved the barrier
- **void** await()
- **void** reset()
- can execute an instance of Runnable when all threads have achieved the barrier

## Exchanger<E>

- a barrier in which a data exchange between threads is done

# Synchronizers

## CyclicBarrier

- blocks threads until a certain number of threads have achieved the barrier
- **void** await()
- **void** reset()
- can execute an instance of Runnable when all threads have achieved the barrier

## Exchanger<E>

- a barrier in which a data exchange between threads is done

# Executor

## Executor

- represents an abstraction allowing to execute a task
- **void** execute(Runnable command)
- more control than with Thread: execution politics, timing etc.

Implementations: ThreadPoolExecutor, ScheduledThreadPoolExecutor to be used with factories from the Executors class.

# Locks

## Lock

- allow to avoid the monitor lock problem
- **void** `lock()`
- **void** `lockInterruptibly()` **throws** `InterruptedException`
- **void** `tryLock()`
- **void** `tryLock(`**long** `to, Unit unit)`
- **void** `unlock()`

## Implementations

- `ReentrantLock`
- `ReentrantReadWriteLock`

# 2 - RTSJ: Real-Time Specification for Java

3. **Real-Time Specification for Java: getting started**

4. **Real-time threads and scheduling**

5. **Asynchronous events**

6. **Memory management**

# Outline of part 2 - RTSJ: Real-Time Specification for Java

3. **Real-Time Specification for Java: getting started**

4. Real-time threads and scheduling

5. Asynchronous events

6. Memory management

# Java SE for real-time?

## Problem with Java SE for RT

- garbage collection
- JIT compiler
- dynamic class loading
- threads management (e.g. priority inversion)

# Exercise: priority inversion example ✎

## Develop three classes

- a `Lock` class with a synchronized method `acquireLock` that make something for 5s
- a `AcquireLockRunnable` class implementing `Runnable` whose `run` method use the `acquireLock` method on a `Lock` object
- a `DummyRunnable` class implementing `Runnable` whose `run` method that make something for 2s

## Create a program

- create a `Lock` object `l`
- create a thread `t1` using `AcquireLockRunnable` on `l` (`MAX_PRIORITY`)
- create a thread `t2` using `AcquireLockRunnable` on `l` (`MIN_PRIORITY`)
- create a thread `t3` using `DummyRunnable` (`NORM_PRIORITY`)
- start `t2`, wait for 5 ms, then `t1`, then `t3` and wait the 3 threads to finish

# Real-Time Specification for Java

The objective is to define a **real-time specification** for Java that solve the previous problems.

The JSR-000001 for a real-time Java specification has been accepted in 1998. Initial PEG (Primary Expert Group) members came from IBM, Aonix/Ada Core, QNX, Sun Microsystems, Rockwell-Collins, Nortel Networks Cyberonics.

Version 1.0.1 of RTSJ was released in 2005. Version 1.1 is planned in JSR 282.

- **JSR 1: Real-time Specification for Java** .
  `http://jcp.org/en/jsr/detail?id=1`.
- **RTSJ** .
  `http://www.rtsj.org/`.
- **JSR 282: RTSJ version 1.1** .
  `http://jcp.org/en/jsr/detail?id=282`.

# Some RTSJ implementations

## Some implementations. . .

- reference implementation: `http://rtsj.org`
- SUN/Oracle Java RTS: `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-139921.html`, available for evaluation
- **aicas JamaicaVM**: `http://aicas.com/sites/jamaica.html`
- FijiVM: `http://www.fiji-systems.com/` (no news at this time. . . )

Beware, some "real-time" JVM do not respect the RTSJ, e.g. JRockit from Oracle (`http://www.oracle.com/technetwork/middleware/jrockit/overview/index-086343.html`) which use a predictible GC but lacks parts of RTSJ.

# Uses of RTSJ implementations

## Some examples

- Eglin Space Surveillance Radar (AN/FPS-85)
- Aonix PERC VM for Aegis Weapon System (but **not RTSJ!**)
- latency-critical banking applications
- ...

# So, what is in RTSJ?

- real-time scheduling
- advanced memory management
- high precision timers
- asynchronous events
- asynchronous interrupts on threads
- no need for GC (**like in the JLS** ☺)

### The `javax.realtime` package

The `javax.realtime` package contains all classes and interfaces for real-time (cf. `http://www.rtsj.org/specjavadoc/book_index.html`).

# aicas JamaicaVM

We will use aicas JamaicaVM during the lab sessions. JamaicaVM is JVM that can execute applications written for Java SE 6.

JamaicaVM has been designed for real-time and embedded systems and proposes:

- **hard** real-time execution guarantees
- support the RTSJ 1.0.2
- minimal footprint (1MB for VM, compaction of classes, smart linking etc.)
- many supported platforms
- fast execution (compilation in C code)
- **tools**

# JamaicaVM: tools

- `jamaicac` a Java compiler based on Open JDK compiler
- `jamaicavm` a JVM
- `jamaicabuilder` which builds a standalone executable with the Jamaica VM + application
- Jamaica Thread Monitor to monitor real-time behavior of applications

+ support of Eclipse with the Jamaica Eclipse Plugin

In `$JAMAICA/doc`:

aicas GmbH (2013).
**JamaicaVM 6.2 - User Manual**.
https://www.aicas.com/cms/sites/default/files/jamaicavm_6.2_manual.pdf.

# Verifying your JamaicaVM installation

(See lab session for details)

## Verifiying JamaicaVM installation at ISAE

1. verify that the env. var. `JAMAICA` is correctly defined
2. start the aicas License Provider to verify it can contact aicas server
3. copy the "Hello world" example in Jamaica distribution and execute it

# Outline of part 2 - RTSJ: Real-Time Specification for Java

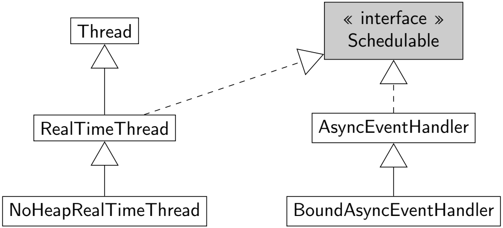
3 Real-Time Specification for Java: getting started

4 **Real-time threads and scheduling**

5 Asynchronous events

6 Memory management

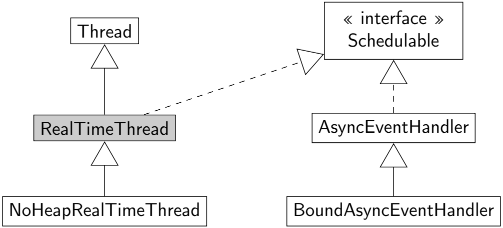
# Available threads types



## Schedulable

- represents objetcs that can be executed by a scheduler
- **really difficult to implement!**

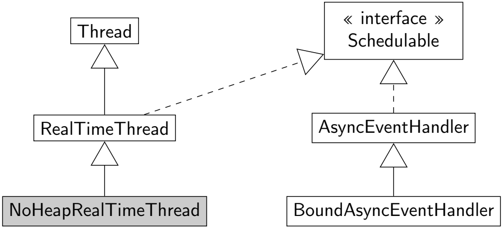
# Available threads types



## RealTimeThread

- extends `Thread`
- access to RT services: asynchronous control transfer, memory, schedulers

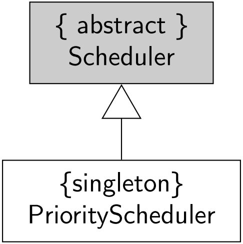
# Available threads types



## NoHeapRealTimeThread

- extends `RealTimeThread`
- is not allowed to allocate or reference object on heap
- **can preempt any GC**
- cf. `ScopedMemory` et `ImmortalMemory`

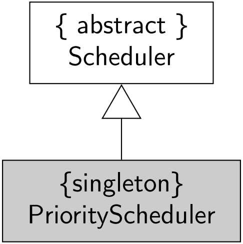
# Scheduling



**Scheduler**

- represents a scheduler for instances of `Schedulable`
- implements a **feasibility algorithm**

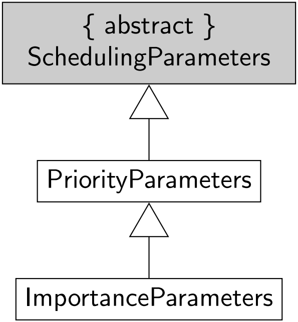
# Scheduling



**PriorityScheduler**

- the only one defined in the specification
- it is a real-time SCHED_FIFO POSIX scheduler
- use **static** `PriorityScheduler` **instance**() to obtain it

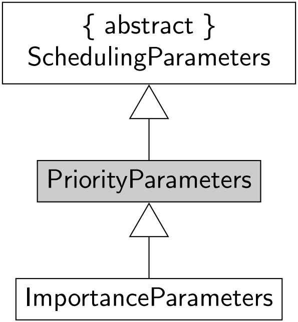
# Scheduling parameters



**SchedulingParameters**

- implements only `clone`

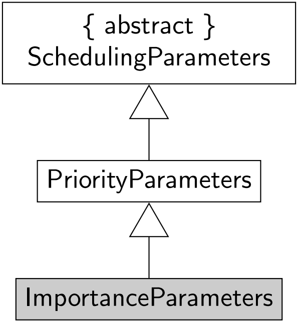
# Scheduling parameters



**PriorityParameters**

- **int** getPriority()
- **void** setPriority(**int** p)
- String toString()

# Scheduling parameters



**ImportanceParameters**

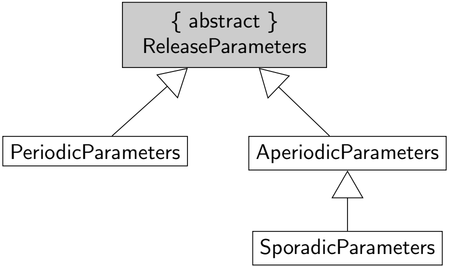
- **int** getImportance()
- **void** setImportance(**int** i)
- String toString()

# Exercise: priority inversion

### Use RT classes!

Use now real-time threads with the priority inversion problem and Jamaica tools set and verify that SCHED_FIFO priorities are respected.

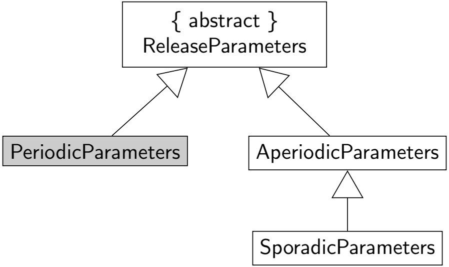
# Release parameters



## `ReleaseParameters`

- associate a `Schedulable` to release characteristics
- cost, deadlines
- *handlers* available when cost is too important or the deadline is missed

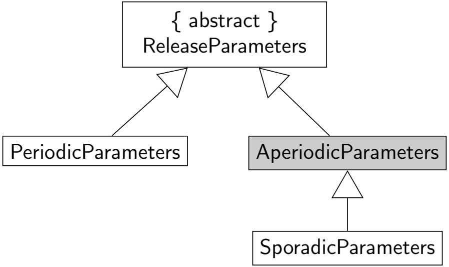
# Release parameters



## PeriodicParameters

- for periodic releases, with an eventual starting delay
- in a `RealtimeThread` instance, use `waitForNextPeriod` method

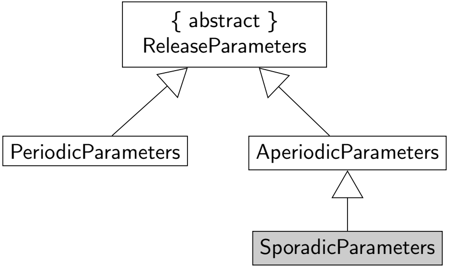
# Release parameters



**AperiodicParameters**

- for releases that can be aperiodic
- triggered using messages for instance

# Release parameters



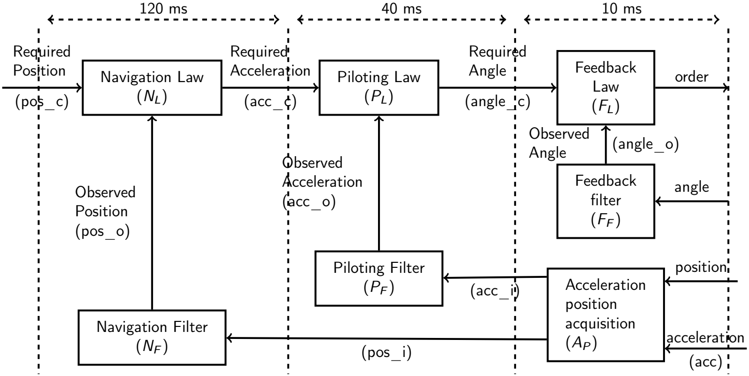
**SporadicParameters**

- like aperiodic, but with a minimum time between releases

# Exercise: flight control system

✎

Thanks to Éric Noulard & Claire Pagetti for this example.



# Outline of part 2 - RTSJ: Real-Time Specification for Java

3. **Real-Time Specification for Java: getting started**

4. **Real-time threads and scheduling**

5. **Asynchronous events**
   - Time triggering
   - Fault triggering and software event triggering
   - Deadline and overrun handlers

6. **Memory management**

# Why asynchronous events?

When dealing with real-time systems, external events trigger some processing:

- a packet arrives
- someone presses a button
- a thread misses its deadline
- ...

Asynchronous events are managed in RTSJ via two classes:

- `AsyncEvent` whose instances represent the asynchronous events
- `AsyncEventHandler` whose instances represent the processing of a particular asynchronous event

# Async. events: how it basically works

## `AsyncEvent`: basic methods

- can be bound to a **external trigger** for the event
- can be bound to a handler for processing
- can be fired: increment the fire count for associated handlers and start any handlers not active

## `AsyncEventHandler`: lifecycle

1. the runtime starts the execution context that will run the event
2. the handler prepares to handle an event
3. the handler invokes its `handleAsyncEvent` method
4. cleanup processing
5. the runtime stops the execution context and puts it away

# Different types of async. events

In the following, we will see different types of asynchronous events:

- time triggered
- fault triggered
- software event triggered
- deadline miss and overrun triggered

## N.B.

In the following, I will sometimes subclass `RealtimeThread` instead of creating implementations of `Runnable` for lack of space ☺

# Outline of part 2 - RTSJ: Real-Time Specification for Java

3 **Real-Time Specification for Java: getting started**

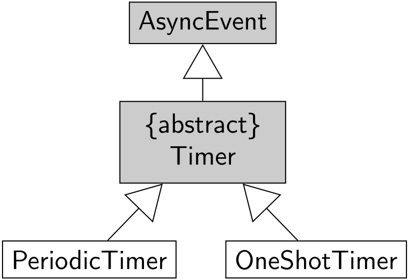
4 **Real-time threads and scheduling**

5 **Asynchronous events**

- Time triggering
- Fault triggering and software event triggering
- Deadline and overrun handlers

6 **Memory management**

# Basic class hierarchy for time-triggered events



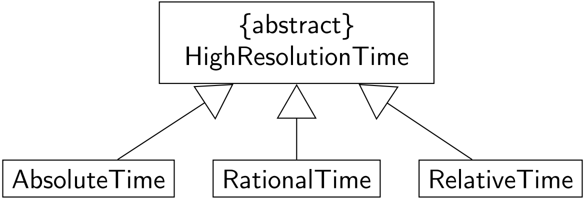
**Timer**

An abstract class for timers. Constructor:

`Timer(HighResolutionTime time, Clock clock, AsyncEventHandler handler)`

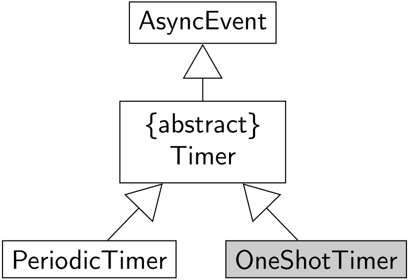
- `time`: the time to fires the event
- `clock`: the reference clock
- `handler`: the handler associated with the timer

# HighResolutionTime hierarchy



- nanosecond accuracy
- `static void waitForObject(Object t, HighResolutionTime t)` can be used as `t.wait(long millis)`
- `RationalTime` is deprecated

# One shot timer



**OneShotTimer**

- execute the associated handlers `handleAsyncEvent` method once at the specified time

# OneShotTimer: watchdog example

**Dog.java**

```java
import javax.realtime.*;

public class Dog {
    static final int TIMEOUT=2000;              // 2 seconds

    public static void main(String [] args){
        double d;
        long n;
        AsyncEventHandler handler = new AsyncEventHandler() {
                public void handleAsyncEvent(){
                    System.err.println("Emergency reset!!!");
                    System.exit(1);
                }
            };

        RelativeTime timeout = new RelativeTime(TIMEOUT, 0);

        OneShotTimer dog = new OneShotTimer(
                             timeout,   // Watchdog interval
                             handler);
```

# OneShotTimer: watchdog example

**Dog.java**

```java
        dog.start();
        while(true){
            d = java.lang.Math.random();
            n = (long)(d * TIMEOUT + 400);
            System.out.println("Running t=" + n);
            try {
                Thread.sleep(n);
            } catch(Exception e){}
            dog.reschedule(timeout);
        }
    }
}
```

# OneShotTimer: timeout example

**OSTimer.java**

```java
import javax.realtime.*;

public class OSTimer {
    static boolean stopLooping = false;

    public static void main(String [] args){
        AsyncEventHandler handler = new AsyncEventHandler() {
                public void handleAsyncEvent(){
                    stopLooping = true;
                }
            };

        OneShotTimer timer = new OneShotTimer(
                                      new RelativeTime(10000, 0),
                                      handler);
```

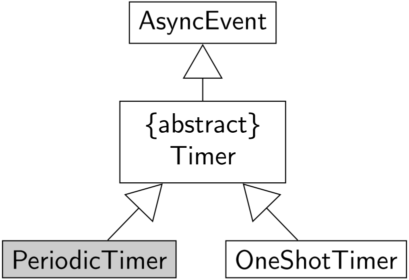
# OneShotTimer: timeout example

**OSTimer.java**

```java
        timer.start();
        while(!stopLooping){
            System.out.println("Running");
            try {
                Thread.sleep(1000);
            } catch(Exception e){}
        }

        System.exit(0);
    }
}
```

# Periodic timer



**PeriodicTimer**

- execute the associated handlers `handleAsyncEvent` method periodically

# PeriodicTimer: example

**PTimer.java**

```
import javax.realtime.*;

public class PTimer {
        public static void main(String [] args){
                AsyncEventHandler handler = new AsyncEventHandler() {
                        public void handleAsyncEvent(){
                                System.out.println("tick");
                        }
                };

                PeriodicTimer timer = new PeriodicTimer(
                        null,      // Start now
                        new RelativeTime(1500, 0),
                        handler);
```

# PeriodicTimer: example

**PTimer.java**

```java
                timer.start();
                try {
                        Thread.sleep(20000);
                } catch(Exception e){ }

                timer.removeHandler(handler);

                System.exit(0);
        }
}
```

# Outline of part 2 - RTSJ: Real-Time Specification for Java

3. Real-Time Specification for Java: getting started

4. Real-time threads and scheduling

5. **Asynchronous events**
   - Time triggering
   - Fault triggering and software event triggering
   - Deadline and overrun handlers

6. Memory management

# Example

**FaultEvt.java**

```java
import javax.realtime.*;

public class FaultEvt extends RealtimeThread{

  static int maxPriority;

  public FaultEvt(SchedulingParameters sched){
    super(sched);
  }
```

# Example

**FaultEvt.java**

```java
  public void run() {
    // Create this method's fault notification
    AsyncEventHandler handler = new AsyncEventHandler() {
      public void handleAsyncEvent(){
        System.err.println("Run method: notified");
      }
    };
    AsyncEvent notify = new AsyncEvent();
    RealtimeThread thisThread =
      RealtimeThread.currentRealtimeThread();

    handler.setSchedulingParameters((SchedulingParameters)
        (new PriorityParameters(maxPriority-3)));

    // Make sure we hear about trouble
    notify.addHandler(handler);
    process1(notify);
    notify.removeHandler(handler);
  }
```

# Example

**FaultEvt.java**

```java
private void process1(AsyncEvent notify){
  //  Create this method’s fault notification
  AsyncEventHandler p2Handler = new AsyncEventHandler() {
    public void handleAsyncEvent(){
      System.err.println("process1 method: notified");
    }
  };
  p2Handler.setSchedulingParameters((SchedulingParameters)
      (new PriorityParameters(maxPriority-4)));
  //  Make sure we hear about trouble
  notify.addHandler(p2Handler);
  process2(notify);
  notify.removeHandler(p2Handler);
}
```

# Example

**FaultEvt.java**

```java
private void process2(AsyncEvent notify){
  //... something bad happened
  //  fire the notification asyncevent.
  {
    notify.fire();
    return;
  }
}
```

# Example

**FaultEvt.java**

```java
  public static void main(String [] args){
    maxPriority = PriorityScheduler.getMaxPriority(null);
    SchedulingParameters sched =
      (SchedulingParameters)(new PriorityParameters(maxPriority-5));
    FaultEvt me = new FaultEvt(sched);
    me.start();
    try{
      me.join();
    } catch (Exception e){};
    System.exit(0);
  }
}
```

# Outline of part 2 - RTSJ: Real-Time Specification for Java

3 **Real-Time Specification for Java: getting started**

4 **Real-time threads and scheduling**

5 **Asynchronous events**
- Time triggering
- Fault triggering and software event triggering
- Deadline and overrun handlers

6 **Memory management**

# What happens with deadline misses and overruns?

Of course, this is for periodic threads.

| Sch. evt | No handler | Handler |
|---|---|---|
| deadline miss | return **`false`** from `waitForNextPeriod` | deschedule at next invocation of `waitForNextPeriod` |
| cost overrun | deschedule immediately and reschedule at next release | deschedule immediately |

# Process for deadline misses

When the scheduler detects that a thread with a miss handler has missed its deadline:

1. it makes the thread nonschedulable
2. it fires its miss handler
3. the thread continues its execution until it invokes `waitForNextPeriod`
4. the thread will **block** until its `schedulePeriodic` method is invoked

# Example: passive miss handler

**PassiveMissHdlr.java**

```
import javax.realtime.*;

/**     Demonstrate a passive miss handler */
public class PassiveMissHdlr {
  /**   Define the passive AEH for misses */
  public static class MissHdlr extends AsyncEventHandler {
    PeriodicThread th;  //  Reference to the client thread

    public void setThread(PeriodicThread th) {
      this.th = th;
    }

    MissHdlr() {
      super(
          new PriorityParameters(
              PriorityScheduler.getMinPriority(null)+11),
          null, null, null, null, null);
    }
```

# Example: passive miss handler

**PassiveMissHdlr.java**

```java
  public void handleAsyncEvent() {
    System.out.println("Recovering from a miss");
    //        First interact with whatever is bothered
    //        by us missing the deadline,
    //  <some action>
    th.schedulePeriodic();  //  Let the thread continue
  }
}
```

# Example: passive miss handler

**PassiveMissHdlr.java**

```java
/**   Define a simple periodic RT thread */
public static class PeriodicThread extends RealtimeThread {
  volatile double f;

  public PeriodicThread(SchedulingParameters sched,
      ReleaseParameters release) {
    super(sched, release);
  }

  public void run() {
    final int CYCLES = 15;
    int bound = 0;

    for (int ctr = 0; ctr < CYCLES; ++ctr) {
      for (f=0.0; f < bound; f += 1.0);  // Use some time
      bound += 800000;
      System.out.println("Ding! " + bound);
      waitForNextPeriod();
    }
  }
}
```

# Example: passive miss handler

**PassiveMissHdlr.java**

```java
  public static void main(String [] args) {
    //  Build parameters for construction of RT thread
    MissHdlr missHdlr = new MissHdlr();
    ReleaseParameters release =
      new PeriodicParameters(
          new RelativeTime(),        // Start at .start()
          new RelativeTime(1000, 0), // 1 second period
          null,                      // cost
          new RelativeTime(500,0),   // deadline=period/2
          null,                      // no overrun handler
          missHdlr);                 // miss handler
    SchedulingParameters scheduling = new PriorityParameters(
        PriorityScheduler.getMinPriority(null)+10);

    PeriodicThread rt= new PeriodicThread(scheduling, release);
    //  Give the miss handler a reference to
    //  the thread it is managing.
    missHdlr.setThread(rt);

    rt.start();   //    Start the periodic thread
    try {
      rt.join();  //    Wait for the thread to end
    } catch (InterruptedException e) {
```

# Outline of part 2 - RTSJ: Real-Time Specification for Java

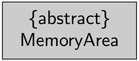
3. Real-Time Specification for Java: getting started

4. Real-time threads and scheduling

5. Asynchronous events

6. **Memory management**

# Memory management: available types

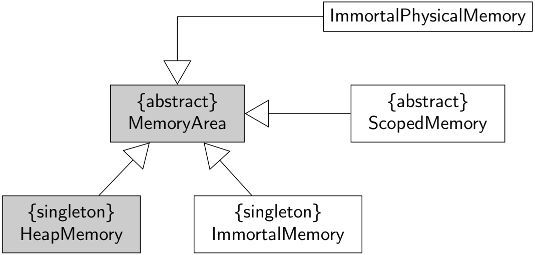


{abstract}
MemoryArea

## MemoryArea

- several constructors (size etc.) + useful methods
- **void** executeInArea(Runnable logic): execute the run methode of logic in this memory area
- **void** enter(): allows a Runnable to enter in this memory area

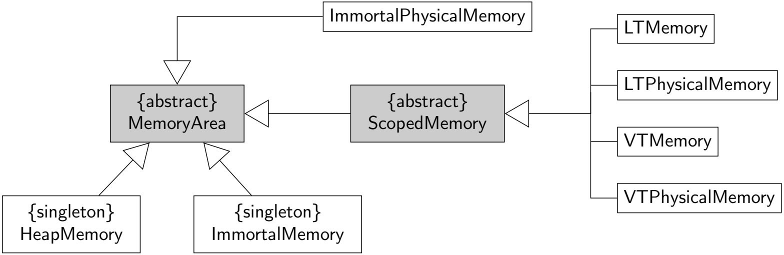
# Memory management: available types



## **HeapMemory**

- the "classical" Java heap. It is accessible via this singleton
- allows to allocate objects on heap even if the thread does not have this context

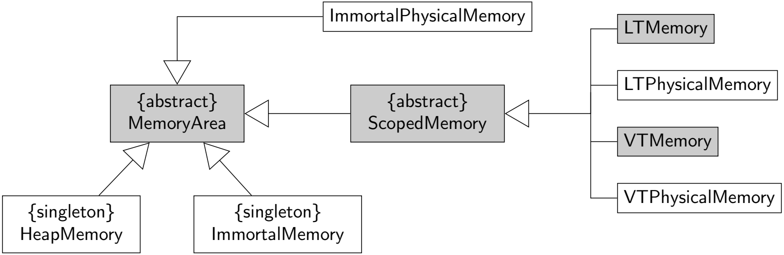
# Memory management: available types



## ScopedMemory

- allows to allocate **dynamically**
- **is not garbage collected**
- objects in this area are finalized when the JVM determines that the scope in no more used by an active thread

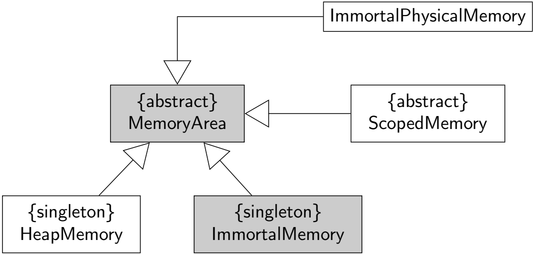
# Memory management: available types



## Several types of `ScopedMemory`

- `LTMemory`: the system guarantees that allocation is in linear time
- `VTMemory`: the system does not guarantee linear time for allocation

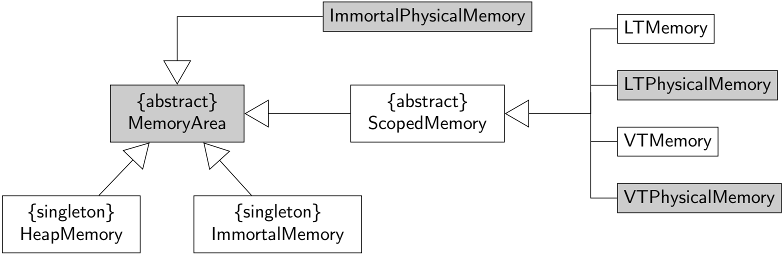
# Memory management: available types



## ImmortalMemory

- a special area for RTSJ accessible via this singleton
- objects in this area exist while JVM is running
- they are accessible from any memory area
- it is never garbage collected!

# Memory management: available types



## Physical memory

- allows to specify a **physical** memory
- constructor: physical address, size, type
- type: DMA, shared memory etc.