

Java Temps Réel

Samia Bouzefrane

Maître de Conférences en Informatique

Laboratoire CEDRIC

CNAM – Paris

<http://cedric.cnam.fr/~bouzefra>

Plan

- Caractéristiques de Java
- RTSJ : Real-Time Specification for Java
- Implémentation
- Bibliographie

Présentation

- A l'origine, Java était prévu pour les équipements embarqués
- Mais Java a rapidement été utilisé pour le Web (applets)
- Puis Java a été utilisé pour le développement d'applications :
 - classiques (J2SE)
 - d'entreprises (J2EE)
 - réduites (J2ME)

Java : quelques avantages

- SDK gratuit
- Java est orienté objet
- Java est portable
- Java est une librairie riche
- La MVJ possède un ramasse-miettes automatique
- Moins sujet à erreur (pas de pointeur comme en C)

Mécanismes de programmation offerts par Java

Java possède :

- Threads
- Moniteurs
- Timers

Threads Java/1

- Les Threads implémentent l'interface `Runnable`, i.e., fournissent la méthode `run()`
- Les threads possèdent des priorités :
de `MIN_PRIORITY (=1)` to `MAX_PRIORITY (=10)`, la priorité par défaut est `NORM_PRIORITY (=5)`.
- Les Threads ont des priorités inférieures à celle du ramasse-miettes
- La MVJ possède un ramasse-miettes automatique
- L'allocation de ressources n'est pas basée sur la priorité

Threads Java/2

- Les Threads démons (daemon)
- Groupe de Threads
- possèdent des méthodes :
`interrupt()`, `yield()`, `sleep()`, `join()`.
- Une Thread peut envoyer une interruption à une autre Thread en exécutant la méthode `interrupt()`
- Une interruption n'est prise en compte par une thread que si celle-ci exécute l'une des méthodes suivantes :
`wait()`, `sleep()`, `join()`, `isInterrupted()`

Démarrer une Thread/1

```
public class MaThread extends Thread {  
    public void run() {  
        System.out.println("Ma nouvelle thread") ;  
    }  
    public static void main(String arg[]) {  
        MaThread tt= new MaThread() ;  
        tt.start() ;  
        System.out.println("La thread mère") ;  
        tt.join() ;  
    }  
}
```

Démarrer une Thread/2

```
public class AutreThread extends Object {  
  
    public static void main(String arg[]) {  
        Runnable objA = new Runnable() {  
            public void run() {  
                System.out.println("Ma nouvelle thread") ;  
            } // fin de run()  
        } ; // fin de objA  
  
        Thread ta= new Thread(objA) ;  
        ta.start() ;  
        System.out.println("La thread mère") ;  
        ta.join() ;  
    } // fin du main  
} // fin de la classe
```

Groupe de Threads

```
public class AutreThread extends Object {  
  
    public static void main(String arg[]) {  
        Runnable objA = new Runnable() {  
            public void run() {  
                System.out.println("Ma nouvelle thread") ;  
            } // fin de run()  
        } ; // fin de objA  
  
        ThreadGroup pere = new ThreadGroup ("Pere") ;  
        Thread ta= new Thread(pere, objA, "ThreadA") ;  
        ta.start() ;  
        System.out.println("La thread mère") ;  
        ta.join() ;  
    } // fin du main  
} // fin de la classe
```

Interrompre une Thread/1

```
public class Interrompre extends Object {  
  
    public static void main(String arg[]) {  
        Runnable objA = new Runnable() {  
            public void run() {  
                System.out.println("ThreadA commence son exécution");  
                try {  
                    Thread.sleep(1000) ;  
                    System.out.println ("ThreadA se réveille apres 1s");  
                } catch (InterruptedException x) {  
                    System.out.println("ThreadA a été interrompue  
pendant son sommeil!");  
                    return ;  
                }  
                System.out.println("ThreadA se termine normalement");  
            } // fin de run()  
        } ; // fin de objA  
    }  
}
```

Interrompre une Thread/2

```
Thread ta = new Thread(objA, "ThreadA") ;  
ta.start() ;  
System.out.println("Thread main envoie une IT a "+ ta.getName());  
ta.interrupt() ;  
  
} // fin du main  
} // fin de la classe
```

Associer une priorité à une Thread/1

```
public class Priorite extends Object {
    public static void main(String arg[]) {
        Runnable objA = new Runnable() {
            public void run() {
                System.out.println (" La Thread " +
                    Thread.currentThread().getName()+ " commence a s'executer ,
                    sa priorite initiale est : " +
                    Thread.currentThread().getPriority()) ;
                try { Thread.sleep(100) ;
                    System.out.println (" La Thread " +
                    Thread.currentThread().getName()+ " se réveille apres 100ms");
                } catch (InterruptedException x) {
```

Associer une priorité à une Thread/2

```
Thread.currentThread().setPriority(7) ;  
  
    System.out.println (" La nouvelle priorite de la  
thread "  
  
    + Thread.currentThread().getName() + " apres  
modification  
  
    par un setPriority est : " + t.getPriority()) ;  
    } // fin de run()  
} ; // fin de objA
```

```
Thread ta= new Thread(objA, " ThreadA ") ;  
  
    ta.start() ;  
  
    ta.join();  
  
} // fin du main
```

```
} // fin de la classe  
samia.bouzeffran@cnam.fr
```

Les moniteurs Java

- un moniteur est un objet qui contient des méthodes accessibles en exclusion mutuelle (`synchronized`)
- Dans un moniteur, on utilise
 - `wait()` pour se bloquer si une condition n'est pas vérifiée et
 - `notify()` ou `notifyAll()` pour réveiller une thread bloquée lorsque la condition devient vraie

Les moniteurs Java : Prod /Cons

```
class ProdConsMonitor {
    private int NbPleins=0, tete=0, queue=0 ;
    private int N ;
    private int tampon[];

    ProdConsMonitor (int n) {
        N=n ;tampon = new int [N] ;
    }

    public synchronized void Deposer (int m)
        throws InterruptedException {
        while (NbPleins == N) { wait(); }
        tampon[queue] =m ; queue = (queue +1)% N;NbPleins ++;
        System.out.println(Thread.currentThread().getName() +
" vient de produire " + m) ;
        notifyAll();
    }
}
```

Les moniteurs Java : Prod/Cons

```
public synchronized int Prelever () throws InterruptedException {  
    while (NbPleins == 0 ) {wait(); }  
    int m= tampon[tete];  
    tete = (tete + 1)% N;  
    NbPleins --;  
    notifyAll();  
    System.out.println(Thread.currentThread().getName() +  
" vient de consommer " + m) ;  
    return m ;  
}  
} // fin de ProdConsMonitor
```

Appel d'un moniteur Java

```
class Producteur extends Thread {
    ProdConsMonitor Mo;

    Producteur ( ProdConsMonitor Mo ) {
        this.Mo=Mo; }

    public void run() {
        try {
            while (true) {
                int m= (int)(1000*Math.random());
                Mo.Deposer(m) ;
                Thread.sleep((int)(1000*Math.random()));
            }
        } catch(InterruptedException e) {}
    } // fin run

} // fin de la classe Producteur
```

Les moniteurs Java : Inconvénients pour le temps réel

- L'entrée dans un moniteur n'est pas basée sur la priorité
- Un `wait(x)` peut provoquer une attente qui dépasse le temps spécifié
- Choisir une thread à notifier n'est pas basé sur la priorité :
 - pas de file d'attente triée par rapport à la priorité
- Pas de protocole évitant l'inversion de priorités

La classe `java.util.Timer`

- L'objet `Timer` est un mécanisme offert pour exécuter plusieurs tâches planifiées par une thread exécutée en arrière plan.
- Les tâches peuvent être planifiées pour s'exécuter une seule fois ou plusieurs fois à des instants réguliers.
- Par défaut, la thread `Timer` n'est pas un démon si bien qu'il est nécessaire de terminer le corps d'une tâche par un appel explicite à la méthode `cancel()` afin que cette méthode termine l'exécution de la tâche.

L'objet `Timer`

Deux constructeurs :

- `public Timer(boolean demon)` ; crée une thread `demon` si `demon` est égal à `true`
- `public Timer()` ; crée une thread `Timer` qui n'est pas un démon.

Arrêt d'une thread `Timer` si :

- un appel à la méthode `cancel()` a été fait. En particulier cet appel peut être fait dans le corps d'une tâche,
- si un appel à la méthode `System.exit()` est fait alors tout le programme se termine.
- la thread `Timer` est un démon. S'il ne reste que des threads démons dans le programme, celui-ci se termine.

`public void cancel()` : termine la thread `Timer` et annule les tâches.

Exécution périodique d'une tâche

`public void schedule(TimerTask tache, long delai) :`
planifie l'exécution de la tâche tache après le délai.

`public void schedule(TimerTask tache, Date instant) :`
planifie l'exécution de la tâche à la date spécifiée.

`public void schedule(TimeTask tache, long delai,
long periode) :` exécution périodique à partir du délai spécifié.

`public void schedule(TimerTask, Date PremierInstant,
long periode) :` exécution périodique à partir de l'instant spécifié.

`public void scheduleAtFixedRate(TimerTask, long delai,
long periode) :` Le taux d'exécution doit être fixe.

`public void scheduleAtFixedRate(TimerTask, Date
premierInstant, long periode)`

La classe `java.util.TimerTask`

L'objet `TimerTask` modélise une tâche dont une thread `Timer` aura à planifier l'exécution. Le corps d'une tâche doit être spécifié dans une méthode `run()`. Une tâche peut connaître la date la plus récente à laquelle elle a été programmée grâce à la méthode `scheduledExecutionTime()`.

`public abstract void run()` contient les actions que devra effectuer la tâche

`public long scheduledExecutionTime()` retourne l'instant le plus récent de planification de la tâche.

Exemple /1

```
import java.util.Timer;
import java.util.TimerTask;
import java.awt.Toolkit;
import javax.swing.* ;

public class ReveilAvecBeep {
    Toolkit toolkit;
    Timer timer;

    public ReveilImageAvecBeep() {
        toolkit = Toolkit.getDefaultToolkit();
        timer = new Timer();
        timer.schedule(new Tache(),
                       0,           //delai initial
                       1*1000);    //periode en ms
    }
}
```

Exemple /2

```
class Tache extends TimerTask {
    int NbBeeps = 4;

    public void run() {
        if (NbBeeps > 0) {
            toolkit.beep();
            System.out.println("Beep "+NbBeeps);
            NbBeeps--;
        } else {
            toolkit.beep();
            System.out.println("Tache accomplie, SORTIR ! ");
            System.exit(0);
        }
    } // fin de run()
} // fin de Tache

public static void main(String args[]) {
    System.out.println("Ordonnancement d'une tache.");
    new ReveilAvecBeep();
} // fin du main
} // fin de la classe
```

Timer : Inconvénients pour le temps réel

- Pas de garanties temps réel
- Si une tâche `Timer` prend trop de temps, elle peut retarder des exécutions futures
- Comment s'exécuter si pas de délais : `scheduleExecutionTime()`

`public abstract void run()` contient les actions que devra effectuer la tâche

`public long scheduledExecutionTime()` retourne l'instant le plus récent de planification de la tâche.

Java : inconvénients pour le temps réel

- Java est lent
- Le GC est imprévisible et peut bloquer un programme pour un temps indéterminé
- L'ordonnancement des threads n'utilise pas de stratégie du TR
- Les moniteurs peuvent causer l'inversion de priorités et pas de file d'attente triée par ordre de priorité.
- Java étant conçu pour être portable, il ne profite pas des possibilités TR offertes par les systèmes d'exploitation.
- Les Timers ne sont pas précis.

RTSJ : Real-Time Specification for Java

- **Real-Time Specification for Java (RTSJ)** est une proposition pour étendre la MVJ avec des fonctionnalités temps réel.
- Version officielle V1.0 en janvier 2002
- Java Specification Request (JSR-00001) sous Java Community

Process (JCP) : 1999-2001

- Proposé par Real-Time for Java Expert Group (Sun, IBM, Nortel, etc.) (<http://www.rtfj.org>)

RTSJ : Contraintes

- Pas de restriction à une version de Java
- N'importe quelles application Java peut tourner sur une JVM RTSJ
- Basé sur J2SE
- Pas d'extension syntaxique
- Pas de prérequis de puissance ou de matériel
- Exécution temporelle prévisible

RTSJ : Aspects non couverts

- Pas de notion :
 - de réseau
 - d'objets distribués
 - de système embarqué
 - d'interruption
 - d'accès au matériel

RTSJ : Aspects couverts

[Bollella et al. 2000]:

- Ordonnancement des threads
- Gestion mémoire
- Synchronisation et partage de ressources
- Gestion des événements asynchrones
- Transfert de contrôle asynchrone
- Temps et Horloge
- Fonctions Système

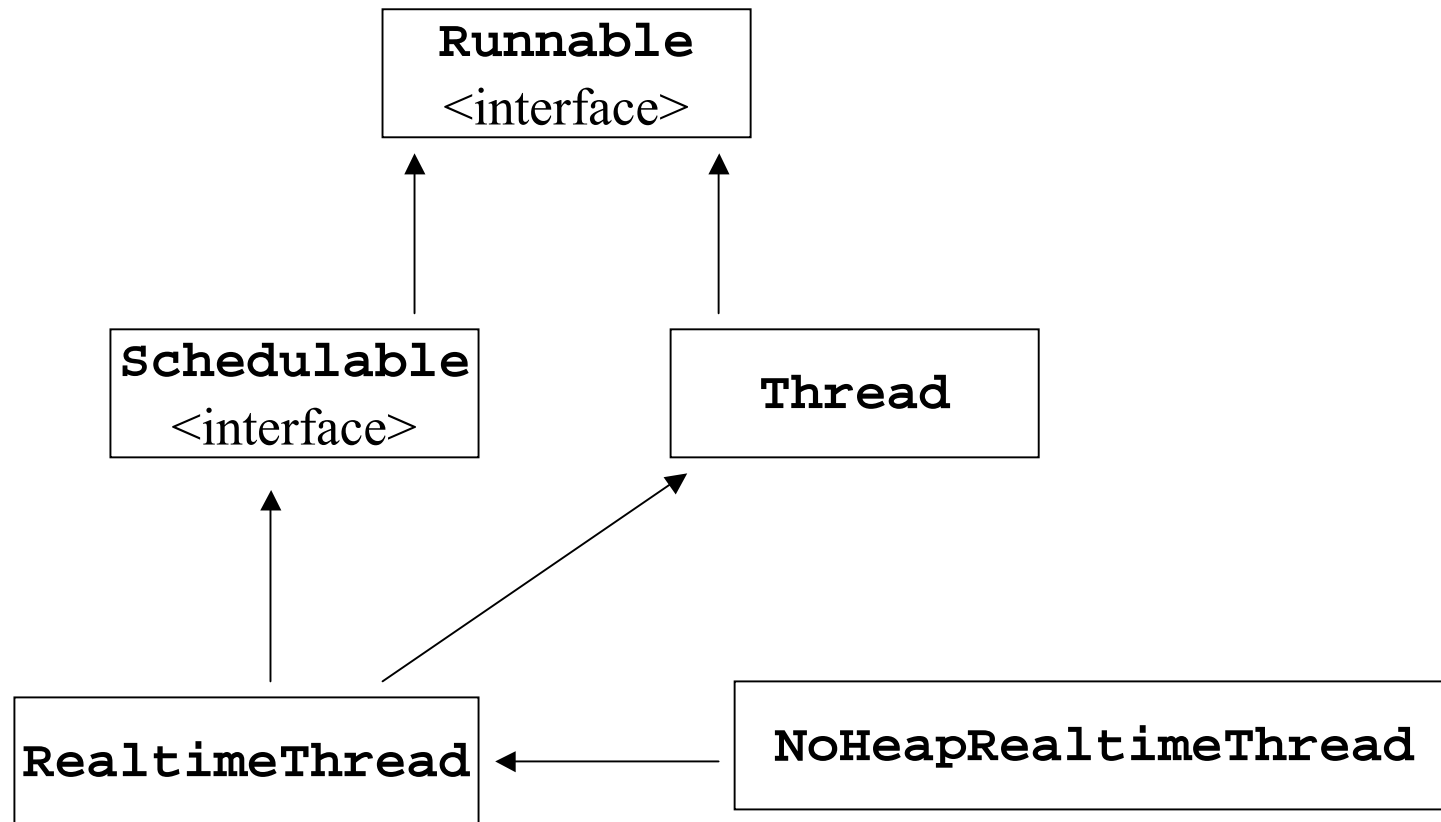
RTSJ : package

- Un package ajouté : `javax.realtime`
 - 3 interfaces
 - 47 classes
 - 13 exceptions
 - 4 erreurs

RTSJ: Threads

- Une interface `Schedulable` : objet pouvant être ordonnancé
- Deux nouvelles classes :
 - `RealtimeThread` étend `java.lang.Thread`
 - peut accéder au tas (heap)
 - de priorité plus petite que celle du GC
 - `NoHeapRealtimeThread` étend `RealtimeThread`
 - pas d'accès au tas (heap)
 - de priorité supérieure à celle du GC
 - travaille dans la mémoire *scoped* ou *immortal*
 - il faut éviter qu'une `NoHeapRealtimeThread` puisse être bloquée par une autre thread qui peut être bloquée par le ramasse-miettes

RealtimeThread



Ordonnanceur de base : PriorityScheduler

- Un seul ordonnanceur est obligatoire :
 - préemptif
 - à priorité fixe
 - au moins 28 priorités
 - ordonnancement FIFO pour le même niveau de priorité
 - le système ne peut pas changer les priorités sauf pour l'inversion de priorités
- RTSJ fournit des outils pour réaliser d'autres ordonnancements (RMS, EDF, LLF)
- La faisabilité de l'ordonnancement est calculable
- Un objet `Schedulable` peut être ordonné

Schedulable / Scheduler

- `Schedulable` Interface qui étend `java.lang.Runnable`
- `Scheduler` est un objet qui
 - Implémente une politique d'ordonnancement basé sur la notion « d'exécution éligible »
- Méthode de contrôle d'admission :

```
boolean addToFeasibility()
```

Schedulable / Scheduler

- `Scheduler.addToFeasibility(Schedulable)` ajoute une tâche dans l'analyse de faisabilité de l'ordonnanceur
- `Scheduler.isFeasible()` pour vérifier si le système est ordonnançable
- changer les paramètres d'une tâche à l'exécution grâce à `setIfFeasible(Schedulable, ReleaseParameters, MemoryParameters)`
- `waitForNextPeriod()` pour les tâches périodiques

Objet Schedulable

Un objet Schedulable utilise les notions de :

- coût (durée d'exécution)
- échéance
- période (pour les tâches périodiques)
- handler de surcharge (*Overrun Handler*, si dépassement de la durée d'exécution)
- handler de faute temporelle (*Miss Handler* si dépassement de l'échéance)

Objet Schedulable

```
setScheduler(  
    Scheduler sched,  
    SchedulingParameters sched_params,  
    ReleaseParameters release_params,  
    MemoryParameters mem_params,  
    ProcessingGroupParameters group)
```

PriorityScheduler est l'ordonnanceur par défaut.

Paramètres d'Ordonnancement/1

RTSJ définit deux types de paramètres d'ordonnancement :

- `PriorityParameters(int priority)`
- un paramètre supplémentaire pour déclarer l'importance des tâches `ImportanceParameters (int priority, int importance)`
- le paramètre d'importance est utilisé par l'ordonnanceur lorsque toutes les tâches ne sont pas ordonnançables dans les temps

Paramètres d'Ordonnancement/2

RTSJ définit les paramètres d'ordonnancement des tâches:

- valeurs utilisées dans l'analyse de faisabilité (WCET, échéance)
- *Overrun handler* (exécuté lors du dépassement du WCET)
- *Deadline miss handler* déclenché lors du dépassement de l'échéance
- `PeriodicParameters` (la thread est exécutée périodiquement)
- `AperiodicParameters` (tâches aperiodiques)
- `SporadicParameters` (tâches aperiodiques avec un intervalle minimal entre deux arrivées successives)

ReleaseParameters

Paramètres temporels - Périodiques

```
PeriodicParameters(  
    HighResolutionTime start, /* si nul lancer au  
démarrage */  
    RelativeTime periode,  
    RelativeTime duree_execution,  
    RelativeTime echeance, /*si nul utiliser la  
période*/  
    AsyncEventHandler overrunHandler,  
    AsyncEventHandler missHandler)
```

ReleaseParameters

Paramètres temporels - Apériodiques

```
AperiodicParameters(  
RelativeTime duree_execution,  
RelativeTime echeance,  
AsyncEventHandler overrunHandler,  
AsyncEventHandler missHandler)
```

Paramètres temporels - Sporadiques

```
SporadicParameters(  
RelativeTime Intervalle_inter_arrivee,  
RelativeTime duree_execution,  
RelativeTime echeance,  
AsyncEventHandler overrunHandler,  
AsyncEventHandler missHandler)
```

ProcessingGroupParameters

Pour un groupe de Threads

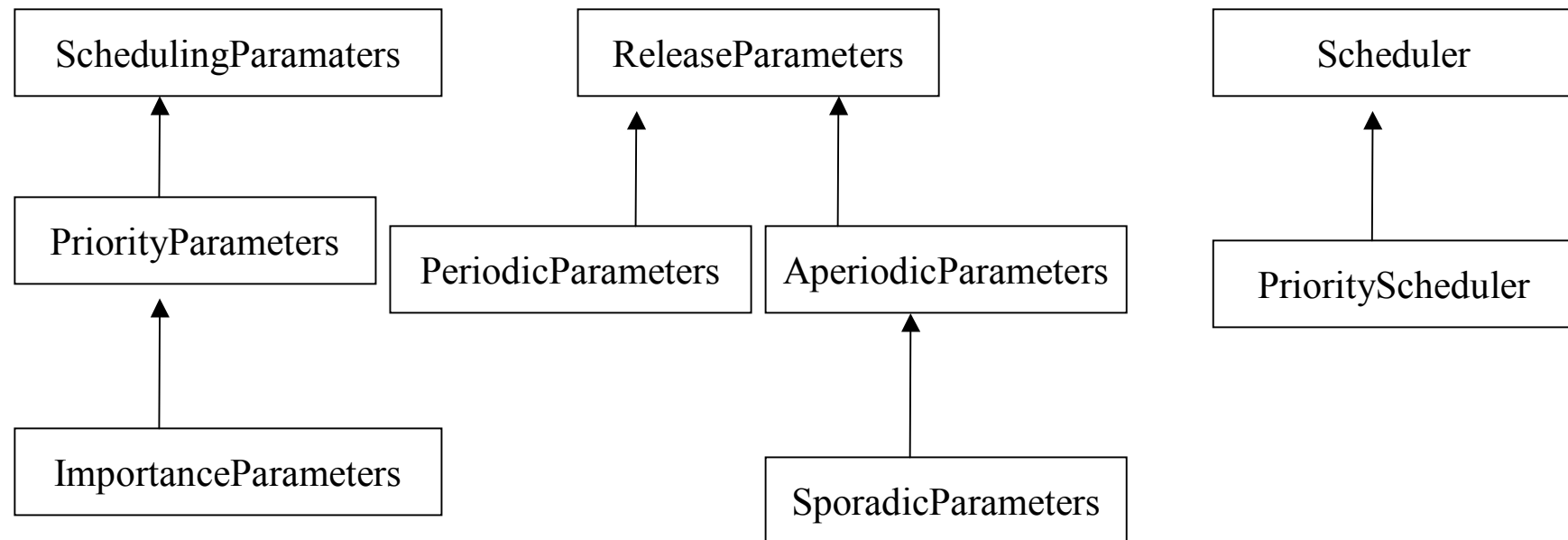
```
ProcessingGroupParameters(  
HighResolutionTime date_reveil,  
RelativeTime periode,  
RelativeTime duree_execution,  
RelativeTime echeance,  
AsyncEventHandler overrunHandler,  
AsyncEventHandler missHandler)
```

MemoryParameters

MemoryParameters

MemoryParameters(
long maxMemoryArea, // ou MemoryParameters.NO_MAX
(unités=octets)
long maxImmortal, //NO_MAX (unités=octets)
long allocationRate) // ou NO_MAX (unités = octets/sec)

Hiérarchie des paramètres d'Ordonnancement



RTSJ : Création d'une tâche périodique/1

```
Scheduler sched = javax.realtime.Scheduler.getDefaultScheduler();
ImportanceParameters prio = new
    ImportanceParameters(MAX_PRIORITY, 3);
PeriodicParameters pp = new PeriodicParameters(
    new RelativeTime(0), // start it when it is released
    new RelativeTime(100), // period
    new RelativeTime(30), // execution duration
    new RelativeTime(60), // deadline
    null, // no Overrun Handler
    null); // no Miss Period Handler
MemoryParameters mp = new MemoryParameters(
    MemoryParameters.NO_MAX,
    MemoryParameters.NO_MAX,
    MemoryParameters.NO_MAX);
MemoryArea ma = new LTMemory(1024, 1024);
ProcessingGroupParameters gp = null;
```

RTSJ : Création d'une tâche périodique/2

```
RealtimeThread rt = new RealtimeThread(prio, pp, mp, ma, gp,  
new Runnable() {  
    public void run() {  
        RealtimeThread t;  
        try {  
            t = (RealtimeThread) Thread.currentThread();  
            do {  
                /* Logique de la Thread */  
            } while(t.waitForNextPeriod());  
        } catch (ClassCastException e) {}  
    }  
});  
rt.setScheduler(sched);  
if (!rt.getScheduler().isFeasible())  
    throw new Exception("Not feasible");  
rt.start();
```

Rate Monotonic Scheduling

L'ordonnancement Rate Monotonic peut facilement être créé à partir de l'ordonnancement `PriorityScheduler` :

- priorités fixes
- périodes connues
- coûts connus

Ramasse-miettes

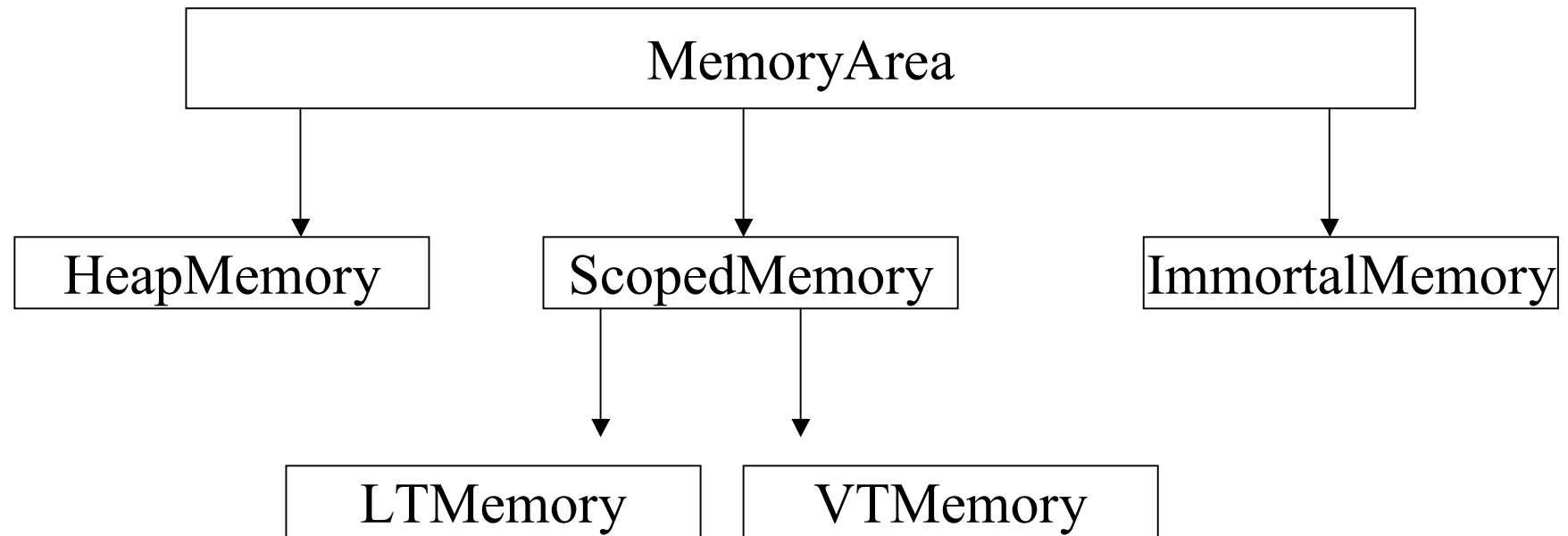
- Problème du ramasse-miettes (GC) traditionnel :
 - le GC peut être exécuté n'importe quand
 - le GC peut durer un certain temps
 - le GC ne peut pas être préempté
 - le GC préempte n'importe quelle thread

- Solutions :
 - ramasse-miettes temps réel
 - RTSJ utilise de la mémoire à portée (`ScopedMemory`)

Gestion de la Mémoire

- Pour Java classique :
 - pile
 - tas où tous les objets sont alloués
- Pour RTSJ :
 - mémoire à portée (`ScopedMemory`)
 - mémoire permanente (`ImmortalMemory`)
 - mémoire à accès direct (`RawMemory`)
 - mémoire physique (`PhysicalMemory`)
 - tas (`HeapMemory`)

RTSJ : Gestion de la Mémoire



Types de zones mémoires

- **HeapMemory** : le tas utilisé par la MVJ
- **ScopedMemory** : a la durée de vie de l'objet qui l'a alloué
- **ImmortalMemory** : a la durée de vie de l'application (espace récupéré lorsque la JVM se termine)
- **PhysicalMemory** :
 - permet un mapping vers des adresses physiques spécifiques
 - peut être de type Immortal ou Scoped.

Types de zones mémoires

	Références vers le tas	Références vers Immortal	Références vers Scoped
Heap	Oui	Oui	Non
Immortal	Oui	Oui	Non
Scoped	Oui	Oui	Oui si la portée est la même ou supérieure

ScopedMemory

- Une ScopedMemory n'est pas gérée par le ramasse-miettes
- Les objets peuvent être alloués dans une ScopedMemory au lieu du tas
- Dès la fin de la portée (scope), les objets sont libérés
- Tous les objets créés dans une thread temps réel peuvent être alloués dans une ScopedMemory
- Les ScopedMemory peuvent être imbriqués

LTMemory : LinearTime Memory

- Une LTMemory est une ScopedMemory
- Le temps d'allocation d'un objet dans une LTMemory est linéaire à la taille de cet objet
- Prédiction du temps d'allocation possible

VTMemory : VariableTime Memory

- Une VTMemory est une ScopedMemory
- Le temps d'allocation d'un objet dans une VTMemory est variable
- Prédiction du temps d'allocation impossible

Utilisation de zones mémoires

- Une zone mémoire peut être liée à une thread temps réel
 - Passé en paramètre
 - Toute allocation est faite dans cette zone
- `void enter(Runnable logic)`
 - exécute `logic` en utilisant cette zone pour l'allocation
- `void enter()`

Accès physique et Accès direct

- Possibilité d'accéder à la mémoire physique
(`PhysicalMemoryManager`)
- Possibilité d'accéder directement à la mémoire, i.e, voir les objets
comme une succession d'octets (`RawMemoryAccess`)
- Ne peut contenir des objets Java

Synchronisation/1

- Problème de l'inversion de priorité :
 - une thread de basse priorité possède un verrou
 - une thread de haute priorité est bloquée lors de l'acquisition de ce verrou
 - une thread de moyenne priorité préempte la thread de basse priorité
 - la thread de haute priorité est bloquée (indirectement) par une thread de moyenne priorité
 - le temps d'acquisition n'est pas borné

Synchronisation/2

- **Solutions :**

- Héritage de priorités (Priority Inheritance)
- Priorité plafond (Priority Ceiling Emulation)

- RTSJ propose que l'héritage de priorité soit l'algorithme par défaut (l'utilisation de ce protocole n'est effective que si le système d'exploitation sous-jacent a implanté ce mécanisme)
- RTSJ offre un outil : les WaitFreeQueues

File de messages sans attente

- Deux types de files de messages sans attente :
 - `WaitFreeWriteQueue`
 - * Lecture bloquante
 - * Ecriture non bloquante
 - `WaitFreeReadQueue`
 - * Lecture non bloquante
 - * Ecriture bloquante
- Exemple: échange de données entre une `NoHeapRealtimeThread` (acquisition) et une `RealtimeThread` (traitement)
NHRT est plus prioritaire que le GC alors que RT est susceptible d'être interrompue par le GC.

Événements Asynchrones

- L'événement : `AsyncEvent` (interruption, signal, etc.)
- Le handler : `AsyncEventHandler`
- L'`AsyncEventHandler` est `Schedulable`
- Lorsque l'`AsyncEvent` est généré, le `AsyncEventHandler` est exécuté
- Un handler peut être associé à plusieurs événements
- Pour réduire le temps de latence : utilisation d'un `BoundAsyncEventHandler`
 - le handler est attaché à une thread.

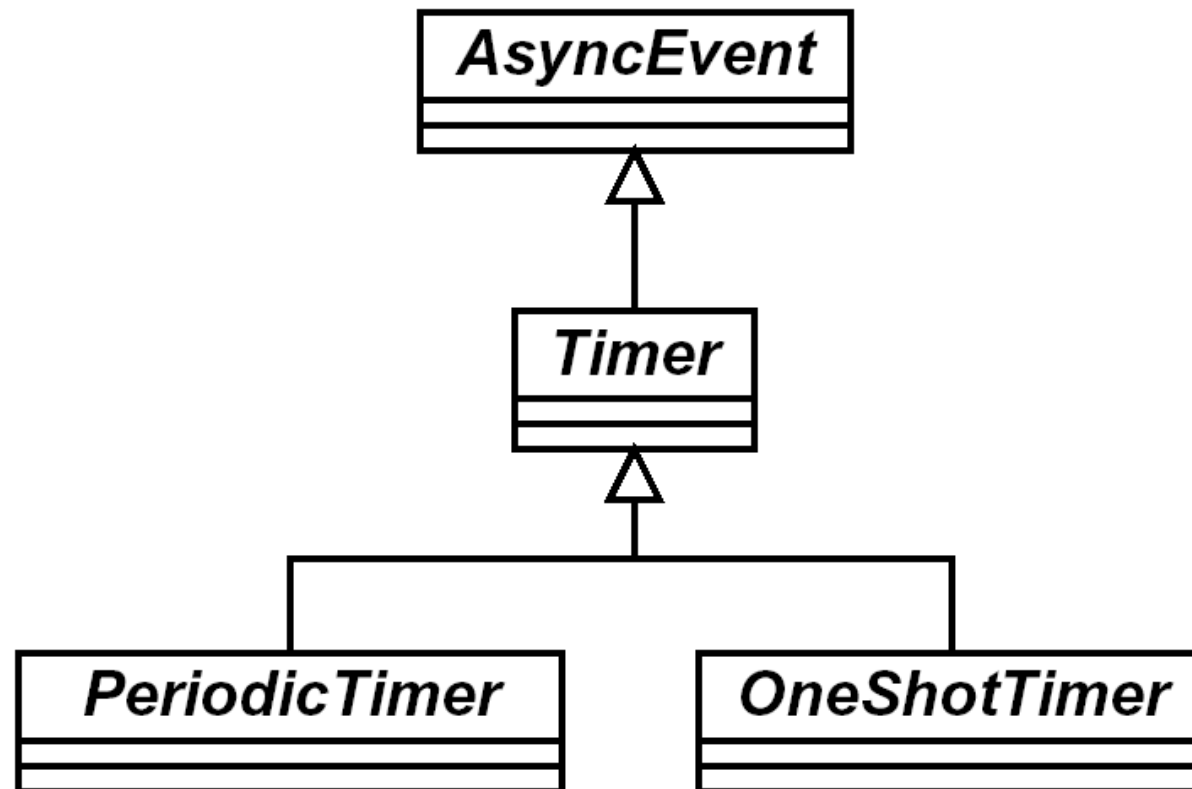
Transfert de contrôle asynchrone

- Principe : contrôler l'exécution d'une thread
 - émission d'une AIE (`AsynchronouslyInterruptedException`)
 - si une méthode capture (`try ... catch`) cette exception, un code particulier est exécuté
- Utile pour borner le temps d'exécution d'une méthode ou contrôler la fin d'une `RealtimeThread`.

Horloges et Timers/1

- L'horloge par défaut est une horloge temps réel
- Un Timer est une forme de `AsyncEvent` relatif à une horloge
- Deux modes :
 - `OneShotTimer` (interruption générée une seule fois)
 - `PeriodicTimer` (interruption générée périodiquement)

Horloges et Timers/2



Les classes `Time`

- Jusqu'à une résolution de nanosecondes :
 - dépend de la résolution du système
 - composants nanoseconde et milliseconde
- Les classes `Time` :
 - `AbsoluteTime` : mesuré depuis le 1er janvier 1970
 - `RealtiveTime` : relatif à une date
 - `RationalTime` : taux d'occurrences par intervalle de temps
exemple: Un Timer périodique a un temps rationnel de $9/250$ s'il est armé 9 fois dans les 250ms.

Les fonctions système

- **PosixSignalHandler** :
 - traite les signaux POSIX (SIGINT, SIGQUIT, etc.) comme des `AsyncEvent`
 - si ces signaux sont fournis par le système.
- **RealtimeSecurity** :
 - pour le contrôle d'accès à la mémoire physique
- **RealtimeSystem** :
 - permet l'accès au GC courant
 - définit le nombre maximal de verrous
 - définit un gestionnaire de sécurité.

Implémentations

- **RI (RTSJ Implementation)** : implémentation de référence
- **jRate** : implémentation opensource
- **JamaicaVM** : implémentation propriétaire

Implémentation de référence

- Implémentation réalisée par la société TimeSys
- Existe uniquement pour Linux/x86
- Prévues pour noyau Linux modifié par TimeSys
- Fonctionne aussi sur noyau standard

jRate

- jRate est un projet universitaire (Université de Californie à Irvine) de Angelo Corsaro
- jRate est une implémentation de RTSJ
- jRate est opensource
- jRate est une extension de GCJ (Gnu Compiler for Java)
- pas très complet (par exemple NoHeapRealtimeThread et file de messages sans attente non implémentés)

JamaicaVM

- JamaicaVM est une implémentation de RTSJ
- est propriétaire (société Aicas)
- possède un ramasse-miettes temps réel

4.5.5 Autres propositions pour Java TR

- RTCE [RTCS 2000] de J-Consortium
- Ravenscar-Java [Kwon et al. 2002]

BIBLIOGRAPHIE/1

[Kloukinas 2003] : *Java & Real-Time : Advantages, Limitations and RTSJ*, Christos Kloukinas, Ecole IN2P3 d'Informatique TR, Verimag Grenoble, 2003.

[Bouzefrane 2003] : LES SYSTEMES D'EXPLOITATION: COURS ET EXERCICES CORRIGES UNIX, LINUX et WINDOWS XP avec C et JAVA, 566 pages, *Dunod Editeur, Octobre 2003*, ISBN : 2 10 007 189 0.

[Bollella 2000] : The Real-Time Specification for Java, Gregory Bollella, transparents disponibles sur <http://www.opengroup.org/rtforum/info/oct2000/slides/rtsj.pdf>, octobre 2000.
The Real-Time Specification for Java, <http://www.rtj.org/rtsj-V1.0.pdf>.

[RTCE 2000]: Real-Time Core Extensions, Real-Time Working Group, Technical Report, J-Consortium, sept. 2000. <http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf>

[Kwon et al. 2002] : J. Kwon, A. Wellings and S. King, Ravenscar-Java : a high integrity profile for real-time Java. In Proc. of the Joint ACM Java Grande – ISCOPE 2002 Conf., pp. 131-140, nov. 2002.

[Cottet 2002] : F. Cottet, J. Delacroix, C. Kaiser, Z. Mammeri, *Scheduling in Real-Time Systems*, J. Wiley & Sons Edition, 2002.

[Burns 2001] : A. Burns and Wellings, *Real-Time systems and Programming Languages : Ada 95, RT Java and RT POSIX*, Addison Wesley, 2001.

BIBLIOGRAPHIE/2

[Cytron et al. 2001] : R. K. Cytron & C. D. Gill, « Real-Time Java », dept. of Computer Science, washigton university in Saint Louis, june 2001.

[Baldo 2004] : Thierry Baldo, « Java Temps Réel », CNAM (Paris), support de cours 2003/2004. <http://jxos.free.fr>

[Holmes 2002]: David Holmes, « Introduction to the Real-Time Specification for Java », Brisbane, Purdue University, Australia, 2002.

~~~~~

JAVA TEMPS RÉEL : [<http://www.j-consortium.org>] [[//www.rtj.org](http://www.rtj.org)]  
[<http://www.real-time.org>]

~~~~~

jRate : <http://www.cs.wustl.edu/~corsaro/jRate>

TimeSys : <http://www.timesys.com>

Aicas (JamaicaVM) : <http://www.aicas.com>

LABORATOIRE CÉDRIC : <http://cedric.cnam.fr>

Site personnel : <http://cedric.cnam.fr/~bouzefra>