# Java Temps Réel

***Samia Bouzefrane***
Maître de Conférences en Informatique
Laboratoire CEDRIC
CNAM – Paris
http://cedric.cnam.fr/~bouzefra

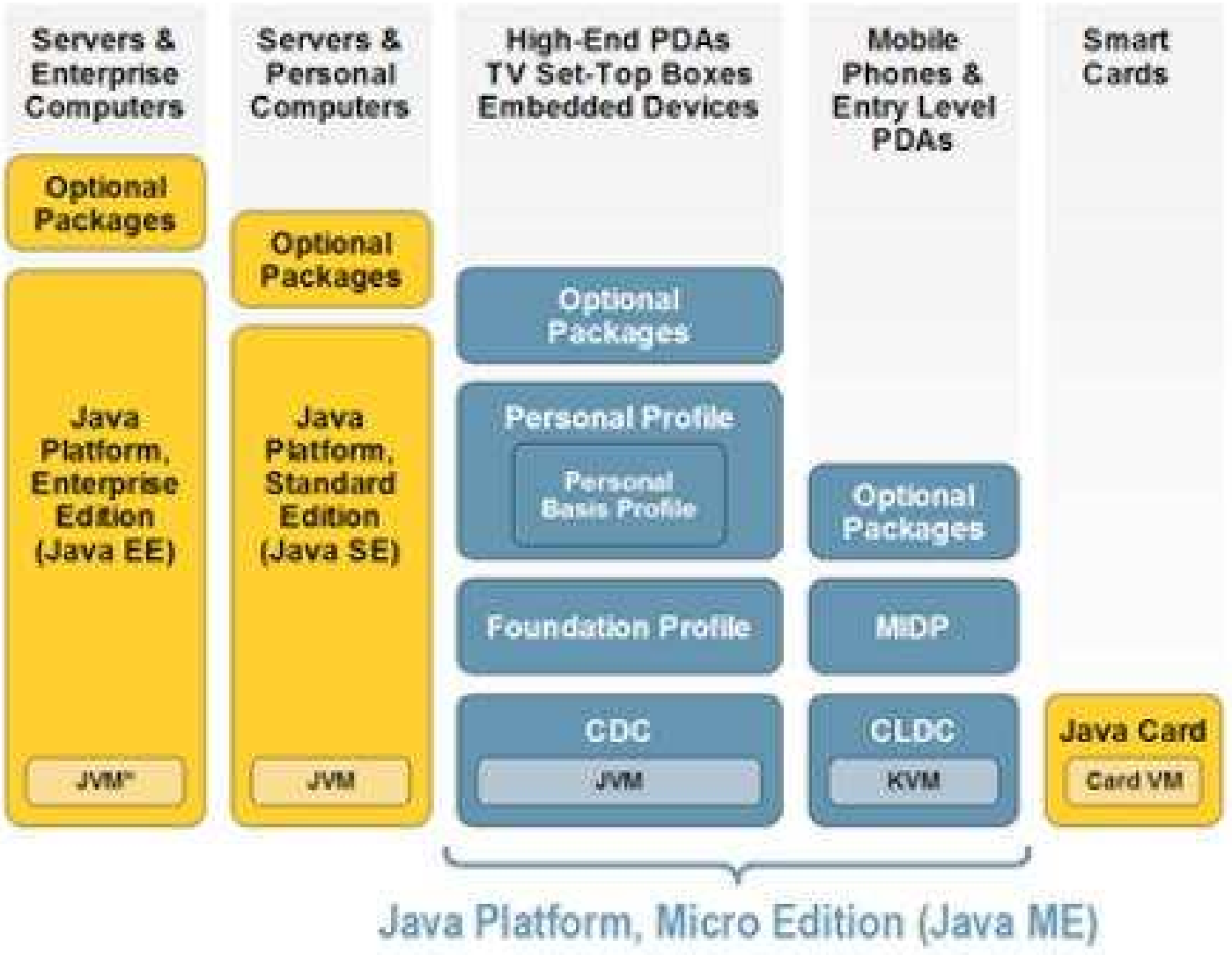
# Sommaire

1. Caractéristiques de Java standard
2. RTSJ : Real-Time Specification for Java
3. Implémentation
4. Références



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Présentation






samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Java : quelques avantages

- **SDK gratuit**
- **Java est orienté objet**
- **Java est portable**
- **Java est une librairie riche**
- **La JVM possède un ramasse-miettes automatique**
- **Moins sujet à erreur (pas de pointeur comme en C)**
- **Java devient de plus en plus performant et fiable grâce à la JVM HotSpot et à l'intégration d'un compilateur JIT (Just in Time)**
  (JIT est un interpréteur basé sur la JVM qui, sous certaines conditions (exécution d'une portion de code plusieurs fois) décide de compiler une partie de l'application)


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Mécanismes de programmation offerts par Java

Java possède des:

- **Threads**
- **Moniteurs**
- **Timers**



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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`**()


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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();
      }
}
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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) { }
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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() ;
   } // fin du main
} fin de la classe
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Les moniteurs Java (version 1.4)

- 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.

(Voir aussi les moniteurs avec les variables condition en Java 1.5 => cours sur la Synchronisation)


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Inconvénients des moniteurs Java 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


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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)`


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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 ReveilAvecBeep() {
        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
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# `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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Java : inconvénients pour le temps réel

- Le GC est imprévisible et peut bloquer un programme pour un temps indéterminé
- L'ordonnancement des threads n'utilise pas de stratégie du temps réel
- 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 temps réel offertes par les systèmes d'exploitation.
- Les Timers ne sont pas précis.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: Real Time Specification for Java

- Real-Time Specification for Java (RTSJ) est une proposition pour étendre la JVM avec des fonctionnalités temps réel.

- Version officielle V1.0 en janvier 2002

- Java Specification Request (JSR-00001) sous Java Community Process (JCP) : 1999-2001

- JSR-000282 (sept. 2005), Version actuelle de RTSJ V1.1

(voir les initiateurs (Sun, IBM, Aonix, TimeSys) de la JSR 282 ainsi que les améliorations apportées: http://jcp.org/en/jsr/detail?id=282)

- Proposé par Real-Time for Java Expert Group (Sun, IBM, Nortel, Orange France, Motorola, Sony Ericsson Mobile Comm., Symbian Ltd, Philips Electronics UK Ltd, Siemens, Intel Corp. etc.)

- (http://www.rtj.org)



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: Contraintes

- Pas de restriction à une version de Java
- Basé sur J2SE
- Pas d'extension syntaxique
- Pas de pré-requis de puissance ou de matériel
- Exécution temporelle prévisible


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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**



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: package

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

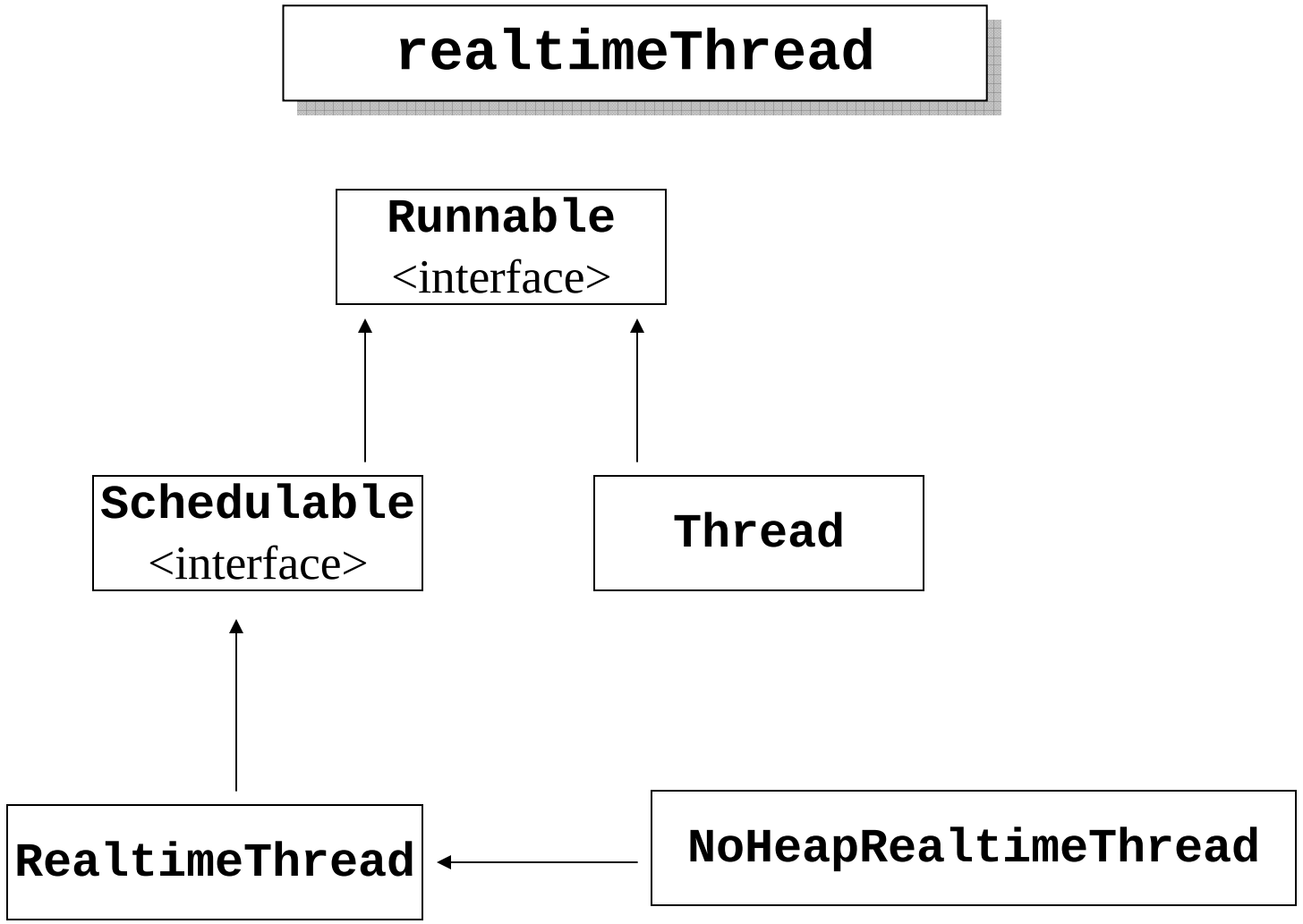


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -


realtimeThread
Runnable
<interface>
Schedulable
<interface>
Thread
RealtimeThread
NoHeapRealtimeThread

# 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 ordonnancé**


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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()
  ```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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)



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Objet Schedulable

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

**PriorityScheduler** est l'ordonnanceur par défaut.



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Paramètres d'ordonnancement

- 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



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Paramètres d'ordonnancement

- 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 apériodiques)
  - `SporadicParameters` (tâches apériodiques avec un intervalle minimal entre deux arrivées successives)


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# ReleaseParameters

➢*Paramètres temporels - Périodiques*

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



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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
)
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# ProcessingGroupParameters

➢ *Pour un groupe de Threads*

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



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# MemoryParameters

- *MemoryParameters*

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


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: création d'une tâche périodique/1

```
import javax.realtime.*;

public class Periodic
{
public static void main (String []arg)
{
PriorityScheduler sched = (PriorityScheduler) javax.realtime.
Scheduler.getDefaultScheduler();

ImportanceParameters prio = new
        ImportanceParameters(sched.getMaxPriority(), 3);

PeriodicParameters pp = new PeriodicParameters(
new RelativeTime(0,0), // start it when it is released
new RelativeTime(100,0),// period
new RelativeTime(30,0), // execution duration
new RelativeTime(60,0), // deadline
null, // no Overrun Handler
null); // no Miss Period Handler
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: création d'une tâche périodique/2

```
MemoryParameters mp = new MemoryParameters(
        MemoryParameters.NO_MAX,
        MemoryParameters.NO_MAX,
        MemoryParameters.NO_MAX);

MemoryArea ma = new LTMemory(1024, 1024);

ProcessingGroupParameters gp = null;

RealtimeThread rt = new RealtimeThread(prio, pp, mp, ma, gp,
new Runnable() {
        public void run() throws Exception {
        RealtimeThread t;
         try {
         t = (RealtimeThread) Thread.currentThread();
         do {
                System.out.println(t.getName() + " en execution ");
           } while(t.waitForNextPeriod());
          } catch (ClassCastException e) {}
       }
  });
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: création d'une tâche périodique/3

```
rt.setScheduler(sched);
if (!rt.getScheduler().isFeasible())
                rt.start();
try {
        rt.join ();
        } catch (InterruptedException e) {   }
 } // fin du main
} // fin de la classe
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Rate Monotonic Scheduling

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

- priorités fixes
- périodes connues
- coûts connus



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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`**)


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

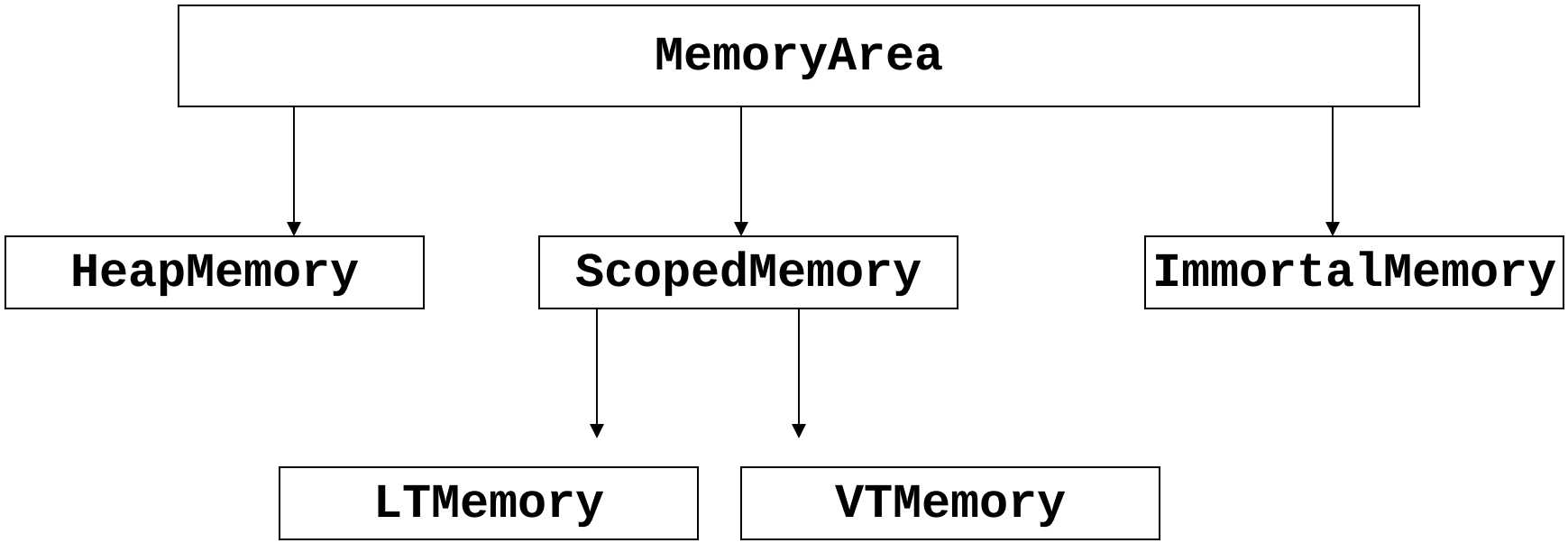
# Gestion de la mémoire

- **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`**)


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RTSJ: Gestion de la mémoire






samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Types de zones mémoire

- **`HeapMemory`** : le tas utilisé par la JVM

- **`ScopedMemory`** : a la durée de vie de la thread temps-réel qui l'occupe

- **`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`**.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Types de zones mémoire

| | 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 |


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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 thread temps-réel occupant la scope, les objets sont libérés
- Les **ScopedMemory** peuvent être imbriquées


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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

- Prévision du temps d'allocation possible



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# VTMemory: VariableTime Memory

- Une **VTMemory** est une **ScopedMemory**
- Le temps d'allocation d'un objet dans une **VTMemory** est variable
- Prévision du temps d'allocation impossible
- Peut être visitée par le ramasse-miettes, contrairement à **LTMemory**


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Utilisation de zones mémoire

- 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()`**


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

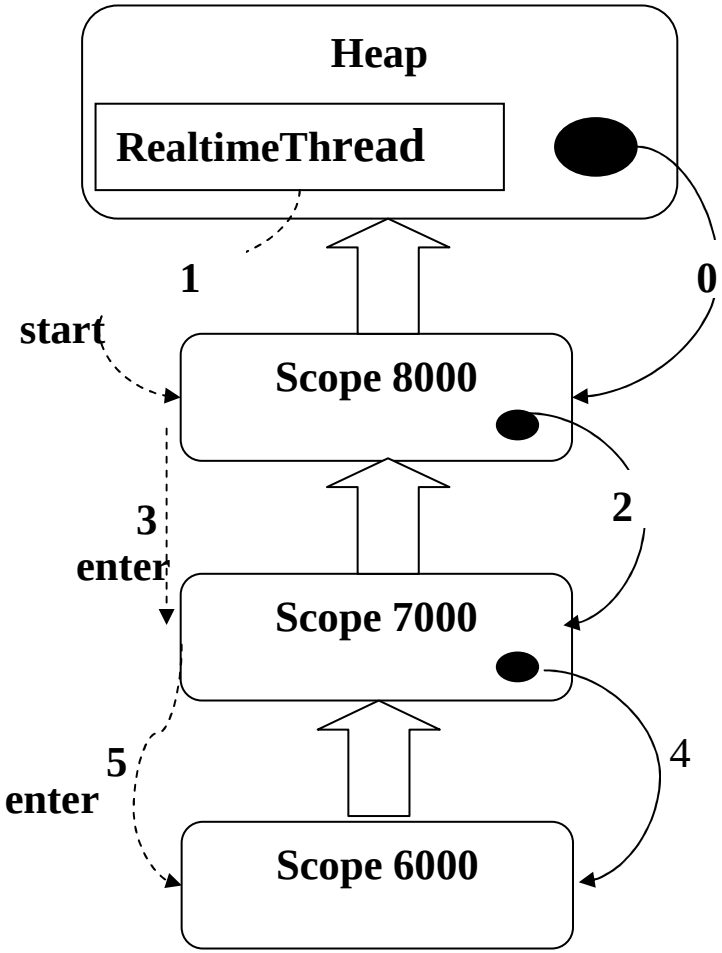
# Exemple d'utilisation de ScopedMemory

```
import javax.realtime.*;
public class ScopeStack extends
RealtimeThread
{   int opt=1;
  public ScopeStack(MemoryArea ma)
        {
super(null,null,null,ma,null,null); }

  public void run() {
  if (opt==1) {
   System.out.println("Memory Area : " +
getCurrentMemoryArea().size());
   LTMemory LT1=new LTMemory(7000,7000);
   opt=2;
   LT1.enter(this);
}
if (opt==2) {LTMemory LT2=new
LTMemory(6000,6000);
   opt=3;
   LT2.enter(this);}
if (opt==3) {
System.out.println("Memory Area
: " +
getCurrentMemoryArea().size());
 }
} // fin run
public static void main(String[]
args)   {
LTMemory LT = new
LTMemory(8000,8000);

ScopeStack ss = new
ScopeStack(LT);
ss.start();
try {LT.join();  }
 catch(Exception ie)    {
    ie.printStackTrace();}
System.out.println("finish!!");
   }
}
```

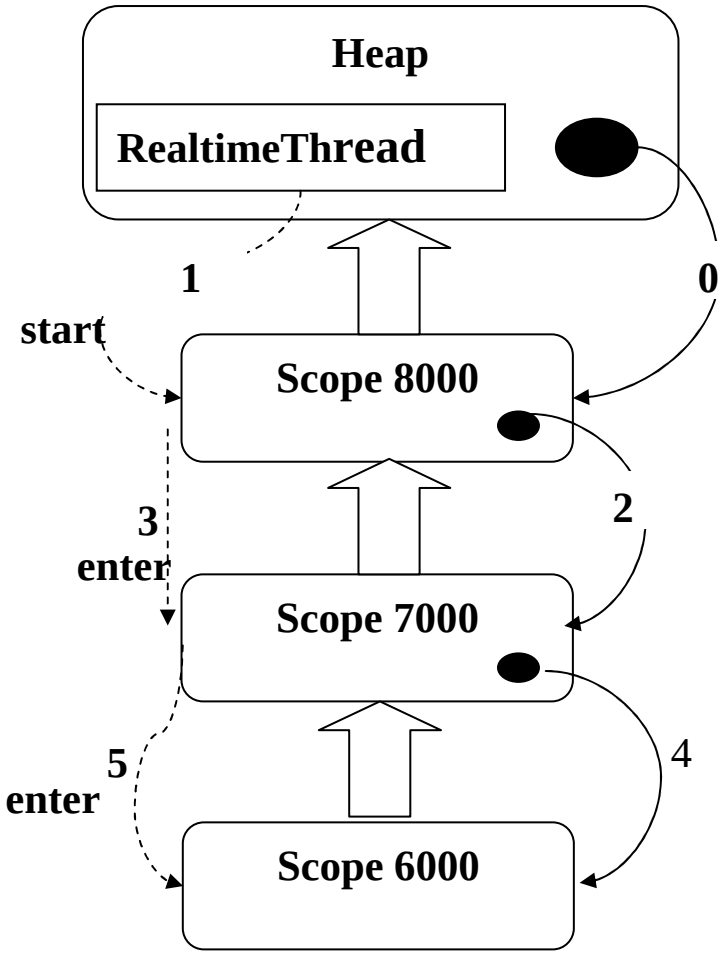


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

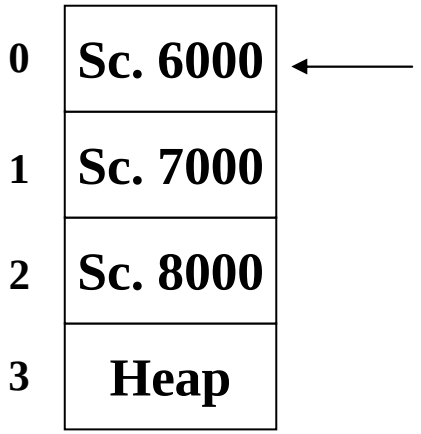
# Exemple d'utilisation de ScopedMemory



# Pile des ScopedMemory




-On peut accéder aux objets de la Scope 8000 à partir de la Scope 7000
-La Scope 8000 ne voit pas les objets de la Scope 7000
-Une thread se trouvant dans une scope donnée peut s'exécuter dans une scope ascendante.




**Pile**



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Fonctions de manipulation de la pile

- **`getMemoryAreaStackDepth`**() retourne la profondeur de la pile (ex. = 3 si 3 zones de mémoires parcourues)

- **`getOuterMemoryArea`** permet l'accès à une scope ascendante se trouvant dans la pile

- **`executeInArea(..)`** permet d'exécuter un objet *Runnable* dans une mémoire scope mère (ascendante)

Exemple : **`getOuterMemoryArea(2).executeInArea(objRunnable);`**



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

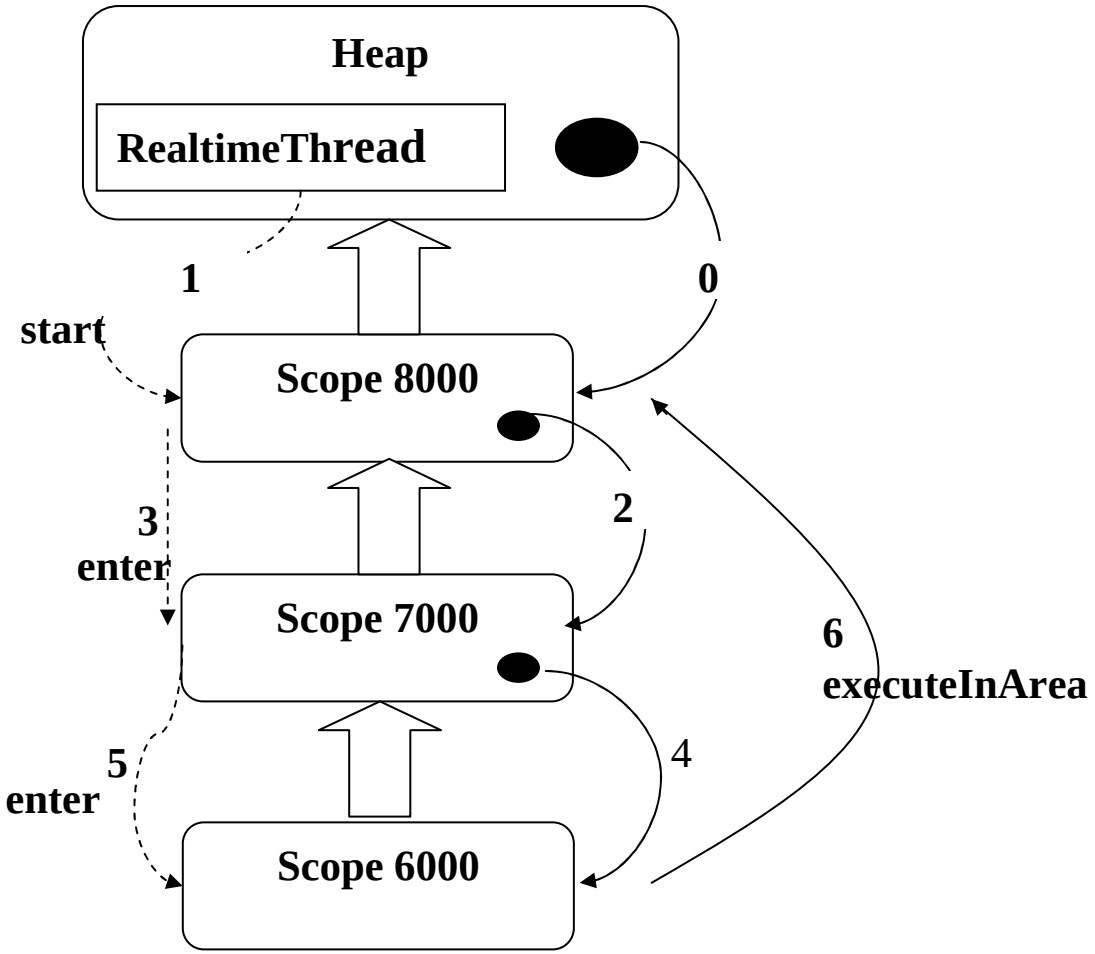
# Exécution dans une autre Scope/1

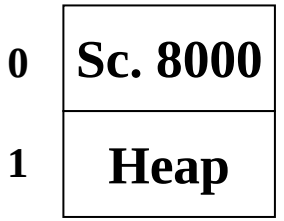
```
import javax.realtime.*;
public class ScopeStack extends
RealtimeThread
{   int opt=1;
  public ScopeStack(MemoryArea ma)
        {
super(null,null,null,ma,null,null); }

  public void run() {
  if (opt==1) {
   System.out.println("Memory Area : " +
getCurrentMemoryArea().size());
   LTMemory LT1=new LTMemory(7000,7000);
   opt=2;
   LT1.enter(this);
}
if (opt==2) {LTMemory LT2=new
LTMemory(6000,6000);
   opt=3;
   LT2.enter(this);}
if (opt==3) {
System.out.println("Memory Area : "
+ getCurrentMemoryArea().size());
getOuterMemoryArea(2).executeInArea
(this);
 }
} // fin run
public static void main(String[]
args)   {
LTMemory LT = new
LTMemory(8000,8000);

ScopeStack ss = new ScopeStack(LT);
ss.start();
try {LT.join();  }
 catch(Exception ie)    {
    ie.printStackTrace();}
System.out.println("finish!!");
   }
}
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Exécution dans une autre Scope/2




**Pile**



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Rendre les objets d'une scope visibles

- Une thread temps réel qui quitte sa scoped memory peut y revenir.
- A son retour, les objets définis au préalable ne lui sont plus accessibles
- **Solution**: déclarer ces objets comme passerelle

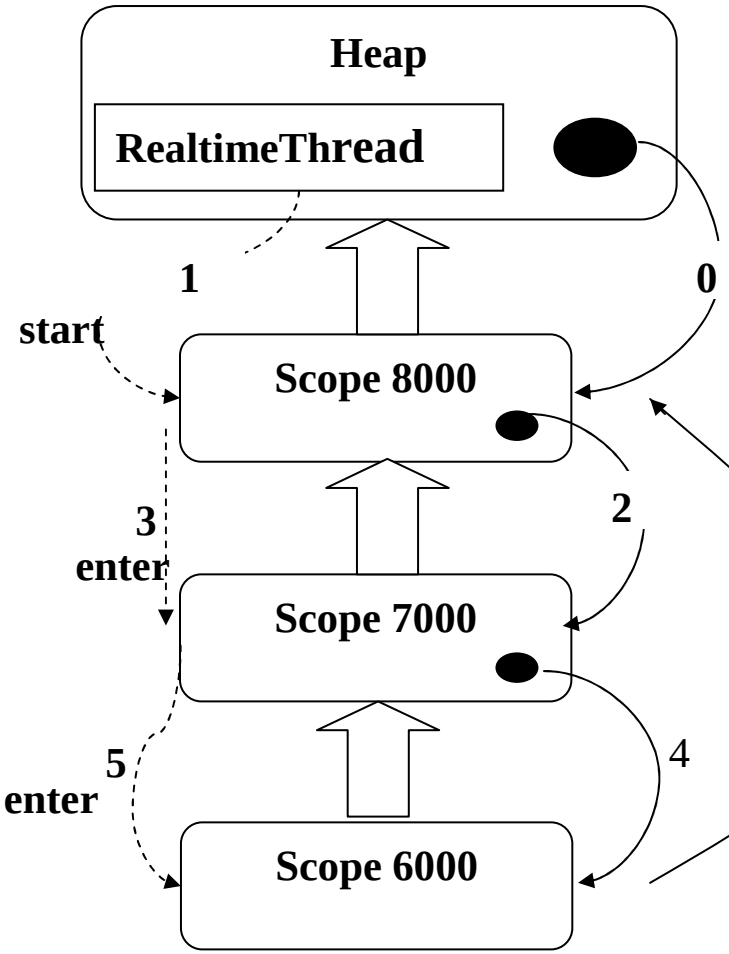
**Exemple** :
Initialement :

```
Integer i = new Integer(7);
((ScopedMemory)getCurrentMemoryArea()).setPortal(i);
```

```
Au retour :

(Integer)((ScopedMemory)getCurrentMemoryArea()).getPortal())
```


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Exemple avec passerelle/1




```
1.Initialement dans la Scope:
 Integer i = new Integer(7);
((ScopedMemory)getCurrentMemoryArea()).
setPortal(i);

2.Au retour dans la Scope :
(Integer)((ScopedMemory)getCurrentMemoryArea()).
getPortal())
```

# Exemple avec passerelle/2

```
import javax.realtime.*;
public class ScopeStack extends
RealtimeThread
{   int opt=1;
  public ScopeStack(MemoryArea ma)
        {
super(null,null,null,ma,null,null); }

  public void run() {
  if (opt==1) {
   System.out.println("Memory Area : " +
getCurrentMemoryArea().size());
Integer i = new Integer(7);
((ScopedMemory)getCurrentMemoryArea()).
setPortal(i);
   LTMemory LT1=new LTMemory(7000,7000);
   opt=2;
   LT1.enter(this);
}
if (opt==2) {LTMemory LT2=new
LTMemory(6000,6000);
   opt=3; LT2.enter(this);}
if (opt==3) {
System.out.println("Memory Area : "
+ getCurrentMemoryArea().size());
Opt=4;
getOuterMemoryArea(2).executeInArea
(this);
 }
if (opt==4) {
Integer j =
   (Integer)((ScopedMemory)getCurr
   entMemoryArea()).getPortal());
   }
} // fin run
public static void main(String[]
args)   {
LTMemory LT = new
LTMemory(8000,8000);
ScopeStack ss = new ScopeStack(LT);
ss.start();
try {LT.join();  }
 catch(Exception ie)    {
    ie.printStackTrace();}
System.out.println("finish!!");
   } }
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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`**


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Files de messages sans attente

- **Deux types de files de messages sans attente :**
  - **`WaitFreeWriteQueue`**
    - * Lecture bloquante
    - * Écriture non bloquante
  - **`WaitFreeReadQueue`**
    - * Lecture non bloquante
    - * Écriture 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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Exemple de gestion d'événements

```
import javax.realtime.*;
public class Handler extends BoundAsyncEventHandler {
        public void handleAsyncEvent() {
          System.out.println("===> Depassement d'echeance ");
          RealtimeThread.currentRealtimeThread().schedulePeriodic();
        }
}
```



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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`**.



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Horloges et Timers

- 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)



samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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

  - **`RelativeTime`** : 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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# 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.


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Implémentations

- **RI (RTSJ Implementation)** : implémentation de référence de TimeSys Corp. (http://www.timesys.com)
- **jRate** : implémentation opensource de l'université de Californie (Irvin) (jRate, http://jrate.sourceforge.net)
- **Ovm**: The Open Virtual Machine Project. http://ovmj.org
- **Mackinac** de Sun MicroSystems: http://research.sun.com/projects/mackinac/ ( Sun's Real-time Java Platform)
- **JamaicaVM** : implémentation propriétaire


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# RI: Implémentation de référence

- Implémentation réalisée par la société *TimeSys Corp.* (http://www.timesys.com)
- Existe uniquement pour Linux/x86 : implémentation gratuite
- Prévue pour le noyau Linux temps réel de *TimeSys* (TimeSys RTLinux/x86)
- L'héritage de priorités n'est implémenté que sur RI/RTLinux de *TimeSys*


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# jRate

- jRate est un projet universitaire (Université de Californie à Irvine) de Angelo Corsaro
- jRate est une implémentation de RTSJ
- jRate est open source
- jRate est une extension de GCJ (Gnu Compiler for Java)
- Tourne sur la famille de machines Linux (Linux/x86 (Red Hat 9, FC2, Debian/unstable, others...), TimeSys RTLinux/x86, TimeSys Linux )


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Ovm

- Projet de l'université de Purdue (USA)
- Ovm est écrit en Java
- Ovm est open source
- Ovm a été utilisé pour la navigation autonome d'un véhicule aérien non-piloté [Prochazka & Vitek 2006]


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# JamaicaVM

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


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Autres propositions pour Java TR

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


samia.bouzefrane@cnam.fr - CEDRIC ( CNAM) -

# Références


**[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.**

**[Prochazka & Vitek 2006]: Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbruster, Edward Pla, David Holmes, "Real-time Java Virtual Machine for Avionics - An Experience Report Jason Baker, Purdue University", 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06) pp. 384-396.**