

***Java
et
le temps réel***

Bertrand Dupouy

ENST

Java et le temps réel

Plan

- **Rappels sur les systèmes temps réel**
- Java et le temps réel : RTSJ
- Mise en œuvre : compilation, paramétrage
- Annexe : JVM embarquées

Contraintes des STR

- Objectifs des STR :
 - déterminisme logique (comme les SE) : les mêmes données en entrées donnent les mêmes résultats en sortie,
 - déterminisme temporel (en plus des SE) : respect des échéances,
 - fiabilité (plus que les SE),

Résultat **correct** = résultat **exact** ... et fourni à la date voulue

- Les bases de temps peuvent être très diverses :
 - radar : millisecondes,
 - BdD : minutes, heures
 - météo, sonde spatiale : heures, jours

Fonctionnement temps réel / Fonctionnement rapide

Récapitulatif : Points critiques

- Ce que doit proposer le logiciel de base (système, langage, machine virtuelle, ...) :
 - Politique d'ordonnancement : ordonnancement approprié -> RMS , EDF, serveur sporadique,
 - Outils de synchronisation,
 - Gestion de la mémoire efficace et déterministe,
 - Précision des horloges,
 - Traitement des événements asynchrones : alarmes, *timers*,
 - Communications déterministes,

JVM : Obstacles au Temps réel : **1-Gestion des threads**

- Pas de politique d'ordonnancement temps réel,
- Spécification imprécise de la gestion des priorités :
 - plusieurs priorités Java peuvent être associées à la même priorité système,
- Le GC, qui est le thread le plus prioritaire, a un comportement non déterministe,
- Inversion de priorité possible avec le mécanisme wait/notify, par exemple :
 - Soit les threads T1, T2 et T3 tels que : Priorité T1 > Priorité T2 > Priorité T3 et une méthode M, `synchronized`, de l'objet O
 - T3 fait O.M(), et rentre dans le code de la méthode M, prenant le verrou associé à O,
 - T1 démarre, donc préempte T3, puis fait O.M() et passe bloqué sur le verrou associé à O,
 - T3 reprend alors la main,
 - T2 démarre, interrompt T3, et prend la main,

JVM : Obstacles au Temps réel : **2-Gestion des timers et des horloges**

- Gestion du temps: java propose Date et Calendar, dont la résolution est de 1 ms,
- Les timers ne respectent pas d'échéances, ils attendent au **moins** le délai spécifié :

```
public class Reveil {
    Timer mon_Timer;

    public Reveil(int Secondes) {
        mon_Timer = new Timer();
        // La tache sera activee dans 5 secondes, au plus tôt
        mon_Timer.schedule(new Tache_Reveil(),Secondes*1000);
    }

    class Tache_Reveil extends TimerTask {
        public void run() {
            System.out.println("Debout!");
            mon_Timer.cancel();
        }
    }

    public static void main(String args[]) {
        new Reveil(5);
        System.out.println("Reveil remonté");
    }
}
```

JVM : Obstacles au Temps réel :
3- Gestion mémoire
Gestion des E/S

- Problèmes dus au garbage collecting :
 - Le GC est automatique, et même s'il est lancé à la demande (System.gc()) , il reste non déterministe,
 - on ne peut pas le désactiver,
- Le temps n'est pas intégré ni dans la gestion des événements, ni dans celle des E/S,

Accès impossible aux adresses physiques en mémoire :

- il faut utiliser le C pour accéder au matériel, écrire des pilotes...

Exemple : La gestion
Des priorités par la JVM

- Les priorités sont numérotées 1 (min) à 10 (max), défaut : 5. Un thread hérite de la priorité de son créateur,
- setpriority() modifie la priorité :
 - paramètres : Thread.MIN_PRIORITY, Thread.MAX_PRIORITY , Thread.NORM_PRIORITY
- Pour setPriority, on trouve :

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority.

Exemple : La classe Timertask

- Trois méthodes de base :

- `public abstract void run();`
- `public boolean cancel();`

- Les méthodes associées à la classe Timer :

- `public abstract void run();`
- pour démarrer la tâche **dès que possible (!)** après Time ou Delay:
 - `void schedule(TimerTask task, Date Time);`
 - `void schedule(TimerTask task, long Delay);`

Plan

- Introduction aux systèmes temps réel
- **Java et le temps réel : RTSJ**
- Mise en œuvre : compilation, paramétrage
- Annexe : JVM embarquées

RTSJ

- Une réponse : RTSJ, *real-time specification for java*, qui est une proposition de fonctionnalités temps réel pour les JVM,
 - implémentation par Sun :RTS,
 - lien RTSJ : <http://www.rtsj.org>
 - lien RTS : <http://java.sun.com/javase/technologies/realtime>
- Un nouveau « package » : `javax.realtime`
 - 3 interfaces, 47 classes, 13 exceptions
- Les applications java « classiques » fonctionnent sur une JVM RTSJ,
- Pas d'extension syntaxique,
- Pas de prérequis sur le matériel,
- Nouveaux types de threads : *realtime thread* et *noheap realtime thread*,
- Gestion des événements (happenings) par des *asynchronous event handlers*,
- Gestion fine du temps : *AbsoluteTime* et *RelativeTime*,
- Ecriture possible de *drivers* en java,

RTSJ : Apports

- Prédicibilité des exécutions,
- Possibilité de calculer l'ordonnançabilité d'un jeu de threads,
- Gestion de la mémoire hors GC,
- Ce qui est spécifié :
 - ordonnancement des threads,
 - utilisation, ou non, du GC pour gérer la mémoire,
 - partage des ressources, synchronisation,
 - gestion du temps,
 - gestion des événements asynchrones,

RTSJ : ***faiblesses***

- Pas de spécification du GC :
 - définition de nouveaux types de mémoire qui **évitent** le GC
- Implémentation pas toujours bien précisée :
 - ***write once carefully, run anywhere conditionnaly***
 - remplace *write once execute everywhere*
- Le réseau : pas d'objets répartis dans RTSJ,

RTSJ : ***ordonnancement***

- La politique d'ordonnancement obligatoirement fournie par le PriorityScheduler fonctionne ainsi :
 - préemptif, 28 niveaux de priorités fixes, gestion FIFO à l'intérieur une même priorité,
 - la JVM ne change les priorités des threads qu'en situation d'inversion de priorité,
- RTSJ fournit de quoi construire des ordonnanceurs RMS, EDF ou LLF,

RTSJ : ordonnancement

- Outils d'ordonnancement :
 - classe abstraite Scheduler permet : test d'ordonnançabilité, dispatching, gestion des événements,
 - interface Schedulable (enrichit runnable) : un Scheduler ordonnance des objets implémentant Schedulable,
 - exemple : RealTimeThread, AsyncEventHandler,
 - chaque objet Schedulable possède une référence vers un Scheduler,
- Deux nouvelles classes de threads :
 - Realtime Thread qui étend Thread. Elle utilise le tas, sa priorité est inférieure à celle du GC,
 - NoHeapRealtime qui étend la précédente :
 - N'utilise pas le tas, mais de la mémoire *scoped* ou *immortal*,
 - **priorité > priorité du GC,**
 - **Attention** un thread NoHeapRealtime peut être bloqué par un thread Realtime, qui sera, peut être, lui-même préempté par le GC...

RTSJ : ordonnancement

- L'interface Schedulable étend Runnable en dotant les objets des caractéristiques supplémentaires suivantes :
 - coût, échéance et, éventuellement, période,
 - fonctions à exécuter en cas de dépassement d'échéance ou du temps d'exécution (Overrun Handler et Miss Handler),
- Scheduler implémente plus qu'une politique d'ordonnancement :
 - contrôle d'admission : Scheduler.addToFeasibility(Schedulable) ajoute un thread pour le calcul d'ordonnançabilité,
 - ce calcul est fait par Scheduler.isFeasible(),
- On peut changer dynamiquement les caractéristiques d'un thread :
 - setIfFeasible (Schedulable, ReleaseParameters, MemoryParameters)
 - waitForNextPeriod() pour les threads périodiques

RTSJ : **Choix de l'ordonnement**

- Classes abstraites :
 - `SchedulingParameters`, paramètres pour l'ordonnement,
 - `ReleaseParameters`, paramètres pour gérer l'exécution : coût, échéance, ...
 - `ProcessGroupParameters` pour gérer les threads apériodiques,
- Définition de la politique d'ordonnement pour un thread :

```
setScheduler (Scheduler Ordonnanceur,  
              SchedulingParameter Ordonnement,  
              ReleaseParameters   Cout,  
              MemoryParameters   Memoire,  
              ProcessingGroupParameters Groupe)
```
- Deux paramètres sont utilisés pour gérer l'ordonnement :
 - la priorité : `PriorityParameters(int priority)`,
 - la criticité, (`ImportanceParameters(int importance)`), utilisée si tous les threads ne sont pas ordonnançables,

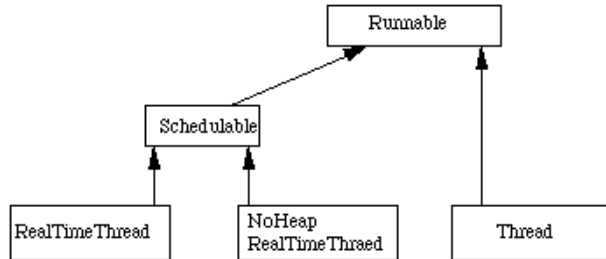
RTSJ : **ordonnançabilité**

- On peut vérifier qu'un ensemble de threads est ordonnançable par une politique donnée.

Exemple très simple : création d'un thread TR et vérification de l'ordonnançabilité :

```
public class Coucou {  
    public static void main(){  
        RealtimeThread Thread_TR_1= new RealtimeThread(){  
            public void run() {  
                System.out.println("Coucou !");  
            } ;  
        };  
        if ( !Thread_TR_1.getScheduler().isFeasible() )  
            System.out.println("Ordonnement impossible.");  
        else  
            Thread_TR_1.start();  
    }  
}
```

RTSJ : ordonnancement



- Exemple de constructeurs pour un thread RT :

```
// Constructeurs possibles :
public RealtimeThread();
public RealtimeThread
(SchedulingParameters scheduling);
public RealtimeThread
(SchedulingParameters scheduling,
ReleaseParameters release);
public RealtimeThread
(SchedulingParameters scheduling,
ReleaseParameters release,
MemoryParameters memory,
MemoryArea area,
ProcessingGroupParameters group,
Runnable logic);
...
}
```

Exemple : paramétrage d'un Thread périodique

- Exemple de définition de la politique d'ordonnancement pour un thread périodique (!!):

```
class Thread_Perio extends RealtimeThread {
...
public Thread_Perio(int Priorite, int Important,
HighResolutionTime Activation,
RelativeTime Periode,
RelativeTime Cout,
RelativeTime Echeance,
int Numero,
MemoryArea Memoire){

super(new ImportanceParameters(Priorite, Important),
new PeriodicParameters(Activation,
Periode, Cout, Echeance,
new OverrunHandler(),
new MissHandler()),
null, Memoire, null, null );

}
}
```

Exemple : Thread périodique

- `waitForNextPeriod` suspend le thread courant jusqu'à l'occurrence de sa période suivante, à moins qu'il n'ait manqué son échéance, quand le thread est relancé cette fonction renvoie `true`.
- `deschedulePeriodic` fera passer le thread à l'état bloqué lors de l'appel à `waitForNextPeriod`. Il reste bloqué jusqu'au prochain appel à `schedulePeriodic`.
- Le scheduler est informé de ces actions et il ajoute, ou retire, le thread de la liste des objets qu'il gère.
- Dans les `ReleaseParameters`, sont éventuellement définis les handlers exécutés si le thread manque une échéance, ou dépasse ses quotas.
 - par défaut, rien n'est fait si une échéance est manquée,
 - si aucun handler n'est spécifié, le nombre d'échéances manquées est incrémenté et sauvegardé.

waitForNextPeriod

- `waitForNextPeriod` suspend le thread courant jusqu'à l'occurrence de sa période suivante, à moins qu'il n'ait manqué son échéance, quand le thread est relancé cette fonction renvoie `true`
- attributs de `waitForNextPeriod` :
 - boolean `lastReturn` — valeur de retour ,
 - integer `missCount` — nombre d'échéances manquées **et pour lesquelles aucun handler n'est défini** ,
 - boolean `descheduled` — bloquer, ou non, le thread à la fin de son activation,
 - integer `pendingReleases` — nombre d'activations pendantes,
- si une échéance est manquée **et qu'un handler lui est associé** :
 - `descheduled` est mis à `true`,
 - le handler est activé avec `fireCount = missCount+1`,
- si une échéance est manquée **et qu'aucun handler ne lui est associé** :
 - `missCount = missCount+1`,
- cet **handler est un objet ordonnançable** :
 - pour qu'il agisse sur le thread avant que ce dernier ne se bloque, il faut que sa priorité soit **supérieure** à celle du thread.

Exemple : Thread périodique

• Création :

```
class Periodique extends RealtimeThread{
    public Periodique (SchedulingParameters SchedPar,
                      ReleaseParameters RelPar){
        super(SchedPar, RelPar);
    }
    public void run(){
        while(true){
            ...
            waitForNextPeriod();
        }
    }
}
```

• Lancement après vérification de l'ordonnançabilité :

```
if (!Periodique.getScheduler().isFeasible())
    System.out.println("Periodique : erreur sur isFeasible");
else
    System.out.println("Periodique peut demarrer");
```

Threads aperiodiques Et sporadiques

- Leurs "release parameters" ne comportent pas de date de départ, ces threads démarrent dès leur lancement (start),
- Pour indiquer leur état (activation terminée, réactivé) : ?
- Pas d'équivalent de `waitForNextPeriod` (cf. les "*sporadic and asynchronous event handlers*" qui disposent de "fire" méthodes)

RTSJ : Paramètres d'ordonnement : Exemple

- Création d'un thread de période 20ms

- Attention : il peut être préempté par le GC !!!!!!!

```
public class Periodique {  
  
    public static void main(String [] args) {  
  
        // Priorite : min+10  
        int Priorite = PriorityScheduler.instance().getMinPriority()+10;  
        PriorityParameters Prio_Par = new PriorityParameters(Priorite);  
  
        // Periode: 20ms  
        RelativeTime Periode = new RelativeTime(20, 0 );  
  
        // Parametres  
        PeriodicParameters Prio_Per = new PeriodicParameters(null,  
                                                             Periode, null,  
                                                             null, null, null);  
  
        // Creation du thread periodique  
        RealtimeThread Thread_Perio = new RealtimeThread(  
            Prio_Par, Prio_Per) {  
  
            public void run() {  
                int n = 0;  
                while (waitForNextPeriod() && (n<100)) {  
                    System.out.println("Coucou : " + n + " fois");  
                    n++;  
                }  
            }  
        };  
  
        // lancer le thread periodique  
        Thread_Perio.start();  
    }  
}
```

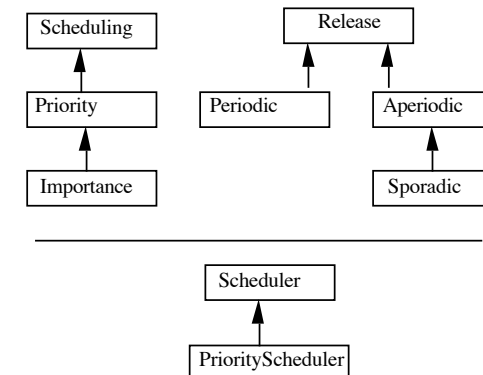
RTSJ : Paramètres d'ordonnement : hiérarchie

- Précisions sur les paramètres d'ordonnement :

- plusieurs types

- PeriodicParameters
- AperiodicParameters
- SporadicParameters (threads aperiodiques avec intervalle minimum entre deux activations)

- Hiérarchie de ces paramètres d'ordonnement :



Paramètres d'ordonnement

- Pour les threads périodiques

PeriodicParameters (HighResolutionTime debut,
RelatriveTime **Periode**, RelatriveTime **Duree**,
RelativeTime **Echeance**,
AsyncEventHandler **DepassementDuree**
AsyncEventHandler **DepassementEcheance**)

- Pour les threads apériodiques :

AperiodicParameters (RelatriveTim **Duree**, RelativeTime **Echeance**,
AsyncEventHandler **DepassementDuree**
AsyncEventHandler **DepassementEcheance**)

- Pour les threads sporadiques :

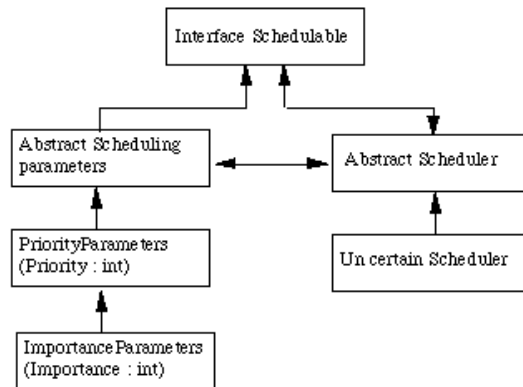
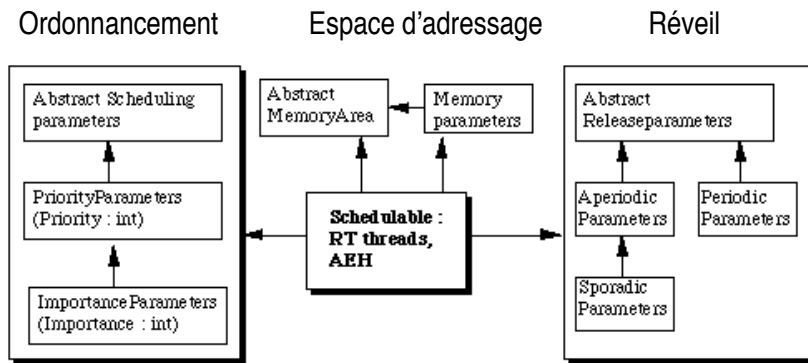
SporadicParameters (RelatriveTime **Intervalle**, RelatriveTime **Duree**,
RelativeTime **Echeance**,
AsyncEventHandler **DepassementDuree**
AsyncEventHandler **DepassementEcheance**)

Récapitulatif Threads

- Les objets ordonnançables ont trois états :
 - bloqué, ne peut passer dans l'état actif :
 - attente fin d'E/S,
 - attente de « reschedule »,
 - attente d'un événement,
 - attente de réinitialisation de « budget »,
 - prêt, peut passer dans l'état actif ,
 - actif : le CO d'un processeur contient une adresse de l'objet,

RTSJ : Ordonnancement

- Schémas récapitulatifs :



RTSJ et Inversion de priorité

- Exemple : échange de données entre T1, un NoHeapRTThread (consommateur), et T2, un RTThread (producteur). On a :

- Priorité(T1) > Priorité(GC) > Priorité(T2),
- T1 va être bloqué par T2 qui peut être interrompu par GC,

- Rappel des deux solutions classiques :

- Héritage de priorité (*Priority Inheritance Protocol*),
- Plafonnement de priorité (*Priority Ceiling Protocol*),

- La RTSJ offre des outils pour gérer PIP : les WaitFreeWriteQueues

- WaitFreeWriteQueues : lecture bloquante, écriture non bloquante,
- WaitFreeReadQueues : écriture bloquante, lecture non bloquante,

Gestion de la mémoire

- Trois types de mémoire :
 - Le tas géré par le GC : HeapMemory ,
 - ImmortalMemory, la mémoire « permanente », qui n'est pas gérée par le GC, elle est rendue à la fin de l'application,
 - Les régions « scoped » qui ne sont pas soumises au GC, sont désallouées automatiquement, elles ont la durée de vie de l'objet qui les utilise:
 - VTMemory, pas de prévision possible du temps d'allocation,
 - LTMemory, temps d'allocation de l'objet prop. à sa taille, prévision du temps d'allocation possible,

Memory		
Heap	Scoped	Immortal
	Ltmemory VTmemory	

- La mémoire physique, PhysicalMemory peut être de type Immortal ou Scoped. Elle sert à « voir » des adresses spécifiques (E/S,...). Ne peut contenir que des suite d'octets (RawMemoryAccess), pas d'objets Java.

Gestion de la mémoire (1)

- Utilisation de la mémoire par les threads :
 - les RT threads peuvent allouer de la mémoire dans les trois régions,
 - les NHRT threads ne peuvent pas allouer, ni même référencer, des objets dans le tas, ils peuvent utiliser la mémoire « immortal » ou « scoped »,
- Contraintes sur les relations entre les différentes régions :
 - **un objet ne doit pas contenir une référence vers un objet dont la durée de vie est inférieure à la sienne,**
 - => le tas et la « immortal memory » ne peuvent contenir des références vers la « scoped memory »,
- Exemple de conséquence :
 - soit une mémoire « scoped » contenant des références vers le tas (permis), un thread NHRT qui utilise cette mémoire (permis) ne pourra pas les lire ces références (il n'a pas le droit d'accéder au tas) !
- Attention : tous les objets `static` sont implantés en IM.

Gestion de la mémoire (2)

- Le tas et la mémoire immortelle :
 - ont chacun une instance unique invocable par instance()
 - implémentent (comme toutes les classes qui héritent de MemoryArea) enter() et executeInArea(),
 - getMemoryArea() permet de savoir où est implanté un objet,
 - les NHRT threads ne peuvent pas allouer, ni même référencer, des objets dans le tas, ils peuvent utiliser la IM et la mémoire scoped,
- Gestion d'une pile « mémoire » : chaque fois qu'un objet **schedulable** entre (sort) dans une zone de mémoire, l'identité de cette zone est empilée (dépilée)
 - un appel à executeInArea sur le tas ou la IM, crée une nouvelle pile

Gestion de la mémoire (3)

- La mémoire scoped :
 - organisation en pile, avec notion de parent : une référence depuis un scope ne peut se faire que vers un scope parent (de niveau supérieur), le tas ou la mémoire immortelle,
 - la parent du premier scope dans lequel entre un thread est le scope « primordial », sinon le scope courant devient le parent du scope où entre le thread, un scope non utilisé n'a pas de parent,
- Notion de parent dans la mémoire scoped (*single parent rule*) :
 - un scope S ne peut avoir qu'un seul parent (le thread qui a fait S.enter si il est en sommet de pile), ou le scope d'où il a été appelé,
 - exemple de cas interdit : lancer deux threads dans deux « scopes » différents et, ensuite, vouloir les faire entrer dans un troisième qui contient des variables partagées :
 - solution-1: lancer les deux threads dans le même scope (donné en zone de mémoire initiale), le parent de ce scope sera le scope primordial
 - solution-2 : lancer les deux thread en IM, puis les faire entrer dans tout de suite dans le scope partagé : comme il est le premier scope dans lequel ils entrent, son parent sera le primordial scope ...

Gestion de la mémoire (4)

- Comptage de référence sur une mémoire scoped :
 - nombre d'appels à `enter`, pas le nombre d'objets qui ont des références vers cette zone de mémoire,

- Règles d'accès à la mémoire :

	Vers le tas	Vers la IM	Vers la mémoire scoped
Depuis le tas	Permis	Permis	Interdit
Depuis la IM	Permis	Permis	Interdit
Depuis la mémoire scoped	Permis	Permis	Permis vers une « précédente »
Depuis une variable locale (sur la pile)	Permis	Permis	Permis

Immortal memory et threads (1)

- Trois techniques pour implanter un NHRT en IM :

1. Passer la IM dans le constructeur :

```
public Thread_TR extends NoHeapRealtimeThread {  
  
public Thread_TR() {  
    super(null, ImmortalMemory.instance());  
    setReleaseParameters(  
        new PeriodicParameters(  
            new RelativeTime(0,0), /* activation */  
            new RelativeTime(1000,0), /* periode */  
            new RelativeTime(5,0), /* C */  
            new RelativeTime(500,0), /* echeance */  
            null, null /* handlers */  
        )); } }  
}
```

Immortal memory et threads (2)

2. On peut utiliser `enter()` en lui passant en argument un objet *runnable*, par exemple :

```
ImmutableMemory.instance().enter(  
    new Runnable(){  
        public void run(){  
            ... ici tout est alloué en immortal  
        }  
    });
```

- Toute la mémoire allouée dans la méthode `run` sera prise en IM. Toute la mémoire allouée par un objet utilisant cette méthode `run` sera allouée au moment du `enter`.

- créer l'objet en IM directement :

```
MemoryArea Mem_Perm = ImmutableMemory.instance()
```

```
ou  
Object O_IM =  
ImmutableMemory.instance().newInstance(Object.class);
```

Scoped memory et threads

- Allocation de *scoped memory* :

```
// Création de la mémoire LT  
Memoire_LT = new LTMemory(20000, 20000);  
  
// création du thread  
Le_Thread = new Thread_Perio  
(TPriorite[i], Timportant, TActivation, TPeriode,  
    TCout, TEcheance, Memoire_LT);  
  
class Thread_Perio extends RealtimeThread {  
    ...  
    public Thread_Perio(int Priorite, int important,  
        HighResolutionTime Activation,  
        RelativeTime Periode,  
        RelativeTime Cout, RelativeTime Echeance,  
        MemoryArea Memoire){  
  
        super(new ImportanceParameters(Priorite, important),  
            new PeriodicParameters(Activation, Periode, Cout,  
                Echeance,  
                new OverrunHandler(),  
                new MissHandler(),  
                null,  
                Memoire, null, null ));  
  
    ...  
    }  
}
```

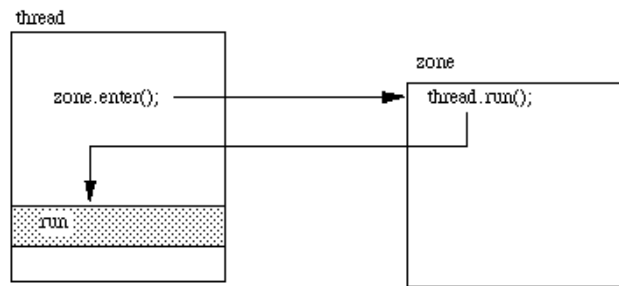
- L'utilisation de cette *scoped memory* par le thread est alors transparente

Scoped memory et threads

- Fonctionnement :

- Tous les threads peuvent accéder à une scoped memory ,
- La JVM gère un compteur de références sur chaque scoped Memory,
- Rôle de `enter(...)` :

`zone.enter(...)` appelle un objet runnable qu'on lui passe en argument, sans argument la méthode `run` du thread courant entre dans la région de mémoire appelée `zone`, par exemple :



Mémoire et threads

- Association d'une zone de mémoire à un thread temps réel :

- En la passant par paramètre,
- Allocation par le thread

- A chaque exécution de `enter (Runnable ...)` ou `enter()` :

- Le compteur de références sur la zone est incrémenté

Exemple :Thread périodique (1)

- Définition des paramètres d'ordonnement :

```
ImportanceParameters Priorite =
    new ImportanceParameters(MAX_PRIORITY, 3);
PeriodicParameters Param_Ordo =
    new PeriodicParameters (
        new RelativeTime(0), // Date de debut
        new RelativeTime(50), // Periode
        new RelativeTime(20), // Duree d'activation
        new RelativeTime(50), // Echeance
        null, null );
```

- Définition des zones de mémoire : allocation dans une Scopedmemory du type LTMemory:

- LTMemory : le temps d'allocation est linéairement proportionnel à la taille de l'objet, la prevision de ce temps est donc possible.
- De plus, une scoped memory n'est pas gérée par le GC : elle a la durée de vie de l'objet

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

```
MemoryArea ma = new LTMemory (1024, 1024);
```

Exemple :Thread périodique (2)

- Le thread temps réel périodique:

```
class Thread_Perio extends RealtimeThread {
public Thread_Perio(int Priorite, int important,
    HighResolutionTime Activation,
    RelativeTime Periode,
    RelativeTime Cout, RelativeTime Echeance,
    MemoryArea Memoire){
    super(new ImportanceParameters(Priorite, important),
        new PeriodicParameters(Activation,
            Periode, Cout, Echeance,
            new OverrunHandler(),
            new MissHandler(Numero)),
        null, Memoire,
        null, null );
}
public void run() {
    System.out.println (Thread.currentThread()
        +"date d activation" +
        Clock.getRealtimeClock().getTime().toString() );
    while (){
        ...
        if(!waitForNextPeriod()){
            ...
        }
    } // fin du while
    ...
} // fin Thread
```

- Lancement après vérification de l'ordonnançabilité :

```
if (Thread_RMS.getScheduler().isFeasible() )
    Thread_RMS.start();
```

Gestion des horloges

- Les bases de temps de la RSTJ :

HighResolutionTime	
RelativeTime	AbsoluteTime
Extends HighResolutionTime	Extends HighResolutionTime
RationalTime	
Extends RelativeTime	

HighResolutionTime : 64 bits pour les ms et 32 bits pour les ns.

- Exemples :

- un PeriodicTimer initialisé avec un RationalTime de (4, 200) sera activé 4 fois par tranche de temps de 200ms, les **intervalles** entre les activations ne sont **pas garantis**
- Prendre le temps absolu en ms :

```
AbsoluteTime date = new  
AbsoluteTime(System.currentTimeMillis(), 0) ;
```

Gestion des timers

- Un timer est un objet piloté par une horloge, deux styles de timer pour la RTSJ :
 - OneShotTimer
 - PeriodicTimer (géré en RelativeTime ou RationalTime)
- A un timer sont associé deux paramètres :
 - date de déclenchement (associé au type d'horloge)
 - fonction à exécuter lorsque l'événement arrive : AsyncEvent Handler, ou AEH, (implémente schedulable)

AsyncEvent	
Associer une horloge	Timer Extends AsyncEvent
PeriodicTimer Extends Timer	OneShotTimer Extends Timer

Gestion des activités périodiques : 1- Thread

```
public Action_Periodique extends RealtimeThread {
    public Action_Periodique() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0), /* activation */
                new RelativeTime(100,0), /* periode */
                new RelativeTime(5,0), /* duree(C) */
                new RelativeTime(50,0), /* echeance */
                null, null /* handlers */
            )
        );
    }
    public void run() {
        while (true) {
            ...
            waitForNextPeriod();
        }
    }
}
```

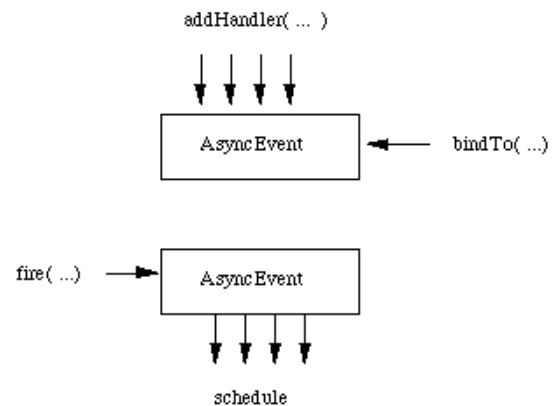
Gestion des activités périodiques : 2- AEH

- Les AEH ont une méthode run, comme les threads.
- Threads ou AEH ?
 - on ne peut pas associer un thread à chaque événement possible !

```
public Action_Periodique extends AsyncEventHandler {
    public Action_Periodique() {
        super();
        setReleaseParameters(
            new PeriodicParameters(
                new RelativeTime(0,0), /* activation
*/
                new RelativeTime(100,0), /* periode */
                new RelativeTime(5,0), /* duree(C) */
                new RelativeTime(50,0), /* echeance */
                null,
                null
            )
        );
    }
    public void handleAsyncEvent () {
        ...
    }
}
```

Gestion des événements Externes et AEH

- Enregistrement des événements à traiter et réponse à leur occurrence :



- Si le même événement déclenche plusieurs handlers, l'ordonnaceur respecte leurs priorités

AEH : Exemple

- On attache un AsyncEventHandler à un AsyncEvent :

```
public static void main(String args)
{
    // Definition d un gestionnaire d evenements (AEH)
    AsyncEventHandler Mon_AEH = new AsyncEventHandler()
    {
        public void handleAsyncEvent()
        {
            do
            {
                System.out.print("Evenement arrivé et traité");
            } while (getAndDecrementPendingFireCount() >0 );
        }
    }

    // Evenement associe a cet AEH
    AsyncEvent Evenement = new AsyncEvent();
    Evenement.addHandler(Mon_AEH);

    // Decelenchement de l evenement
    Evenement.fire();

    System.out.println("L'evenement a été envoyé.\n");
}
```

ÉvénementsAsynchrone ,Timers Classe Time

- On attache un AsyncEventHandler à un AsyncEvent :
 - AsyncEventHandler est « Schedulable »
 - Peut servir à borner le temps d'exécution d'une méthode, vérifier une fin d'un thread, ...
- Un Timer est un AsyncEventHandler attaché à une horloge :
 - interruption émise une fois : OneShotTimer
 - interruption périodique : PeriodicTimer
 - les signaux POSIX sont gérés comme des AsyncEvent (PosixSignalHandler)
- Plusieurs classes Time :
 - AbsoluteTime
 - RelativeTime
 - RationalTime : fréquence,
 - Exemple : un timer a un RationalTime de 4/100 si il est armé 4 fois en 100ms

ATC (Asynchronous transfer control)

- raffinement de **interrupt()** de la JVM classique qui n'est prise en compte que lorsque le thread est bloqué (sleep, join, wait)
- RTSJ permet de dérouter un flux d'exécution grâce aux :
 - AIE(AsynchronouslyInterruptedException) associées aux ATC
- différence entre les différents événements asynchrones :
 - ATC : pour gérer un événement à traiter **immédiatement**. Même style de traitement que celui d'une interruption.
 - AEH (AsyncEventHandler) : pour gérer un événement qui peut être traité comme un thread : on prend en compte son style d'ordonnancement, sa priorité...

ATC : **Fonctionnement**

• Réception :

- les threads TR choisissent de recevoir ou d'ignorer une AIE :
`void run() throws asynchronouslyInterruptedException`

• Emission :

- utilisation de `fire()`, (si la méthode n'est pas spécifiée AIE via une clause « throws », on remonte l'interruption émise par le `fire` jusqu'à ce qu'on atteigne une méthode qui l'a autorisée.
- utilisation d'`interrupt` sur un thread **temps réel**

• Propagation :

- n'intervient pas une section critique
- différences avec `throw/try/catch` de la JVM classique :

Exemple

• Programme:

```
class Mon_thread extends RealtimeThread {
    public void run(){
        System.out.println(" Etape 1 ");
        AIE_Non_1();
        System.out.println(" Etape 2 ");
    }

    public static void main(String[] args){
        Mon_thread Le_Thread = new Mon_thread();
        Le_Thread.start();
        ...
        Le_Thread.interrupt();
        ...
    }

    void AIE_Non_1(){
        try {
            System.out.println(" AIE_Non_1 : Etape 1 ");
            AIE_Oui_1();
            System.out.println(" AIE_Non_1 : Etape 2 ");
        }
        catch(AsynchronouslyInterruptedException aie)
        {
            System.out.println(" AIE_Non_1 Catch AIE ");
            aie.happened(false);
        }
    }
}
```

Exemple

```
void AIE_Non_2(){
    try{
        System.out.println("Entering try in
AIE_Non_2");
        throw new RuntimeException();
    }
    finally {
        System.out.println("In AIE_Non_2, finally");
    }
}

void AIE_Oui_1() throws
AsynchronouslyInterruptedException{
    try {
        System.out.println(" AIE_Oui_1 : Etape 1 ");
        AIE_Oui_2();
        System.out.println(" AIE_Oui_1 : Etape 2 ");
    }
    catch (Exception e){
        System.out.println(" AIE_Oui_1 Catch AIE ");
    }
    finally{
        System.out.println (" AIE_Oui_1 finally");
    }
}

void AIE_Oui_2() throws
AsynchronouslyInterruptedException{
    try{
        System.out.println(" AIE_Oui_2 : Etape 1");
        AIE_Non_2();
        System.out.println(" AIE_Oui_2 : Etape 2");
    }
    catch (Exception e){
        System.out.println("AIE_Oui_2 Catch AIE ");
    }
    finally {
        System.out.println(" AIE_Oui_2 finally");
    } } }
```

Exemple Exécution (1)

• Le résultat dépend de là où en est l'exécution de Le_Thread lors de interrupt(). Premier scénario, affichage :

```
Etape 1
AIE_Non_1 : Etape 1
```

=====

Commentaires :

- 1- Le thread rentre dans run
 - 2- il appelle AIE_Non_1
 - 3- Affichage de : "AIE_Non_1 Etape 1 "
 - 4- fin de son quantum
 - 5- main reprend la main et envoie l'interruption
 - 6- elle est marquée pendante pour Le_Thread
 - 7- AIE_Oui_1 est appelée, on sort du bloc try
 - 8- on entre dans le bloc catch de AIE_Non_1
 - 9- happened met argument à false, donc on revient sans AIE pendante,
- on retourne dans run, fin.

Exemple Exécution (2)

- Second scénario :

```
Etape 1
AIE_Non_1 : Etape 1
AIE_Oui_1 : Etape 1
AIE_Oui_2 : Etape 1
AIE_Non_2 : Etape 1
AIE_Non_2 finally
AIE_Non_1 catch
Etape 2
```

- Commentaires :

1-AIE pendante dans Le_Thread

2- au retour de println, l'execution continue dans AIE_Non_2,

3- RuntimeException émise, le bloc finally est exécuté

4- une AIE générique est remontée vers AIE_Oui_2.

5- Les AIE ne sont pas prises, l'exception est propagée après l'appel à AIE_Oui_1, dans AIE_Non_1

6- prise par catch, happened = false, AIE n'est plus pendante

7- retour à run.

Conclusion

- Attention aux NHRT Threads qui communiquent avec d'autres threads : ceux ci peuvent être interrompus par le GC !
- Java peut-il encore être qualifié de WORA (Write Once Run Anywhere) ?
- Pas d'accès à RMI, pas de gestion des communications réseau.
- Attention : vérifier la gestion PIP/PCP des zones `synchronized`

Plan

- Rappels sur les systèmes temps réel
- Java et le temps réel : RTSJ
- **Mise en œuvre : compilation, paramétrage**
- Annexe : JVM embarquées

Compilation : Mode standard

- Mode standard: JIT (*Just In Time compilation*). La résolution des symboles non définis, etc, est traitée lors de l'exécution.
Fonctionnement :
 - à l'appel d'une méthode, un compteur est incrémenté, quand il atteint un certain seuil, la méthode est compilée,
 - le même mécanisme est mis en œuvre pour les boucles,
 - quand se fait la compilation ? Deux modes :
 - asynchrone (le défaut) : le thread courant n'est pas arrêté, le thread de compilation est exécuté en arrière-plan (*background*),
 - synchrone : le thread courant se bloque en attente de la fin de la compilation,
 - flag : `-XX : -BackgroundCompilation`

Compilation ITC

- Mode ITC (*Initialization Time Compilaion*). La compilation des classes se fait au moment de leur initialisation, c'est-à-dire lors de leur première utilisation.
- On peut intervenir à trois niveaux, en définissant :
 - les classes compilées lors de leur initialisation,
 - les classes qui seront pré-initialisées,
 - les classes qui seront pré-chargées,
- Le plus simple est de le faire définir les classes à pré-compiler par la JVM RTS :

```
### option pour construire la liste de precompilation
### pour de futures executions
-XX:+RTSJBuildCompilationList      (->nhrt.precompile)

### options pour construire la liste de prechargement
### pour de futures executions
-Drtsj.precompile=nhrt.precompile
-XX:+RTSJBuildPreloadList          (->itc.preload)
-XX:+RTSJBuildClassInitializationList (->itc.preinit)

### options pour une execution
-Drtsj.precompile=nhrt.precompile
-Drtsj.preload=itc.preload
-Drtsj.preinit=itc.preinit
```

- Si on spécifie à la fois preload et preinit, alors preload est ignoré parce que preinit implique preload. Attention : ces fichiers ont des contenus différents, donc on peut charger des classes qu'on n'utilisera pas, et par conséquent, elles ne seront pas initialisées.

Compilation : Les threads

- Thread NoHeapRealTime : ITC obligatoire,
- Thread RealTime :
 - ITC par défaut, pour la supprimer : `-XX : -ITCRT`
 - JIT par défaut, pour la supprimer : `-XX : -JITRT`
 - Si on est en ITC, les méthodes contenues dans la liste à précompiler sont compilées à l'initialisation des classes, les **autres** sont interprétées (`-XX : -JITRT`) ou JIT'ées (`-XX : +JITRT`)
 - La compilation asynchrone est le mode par défaut en mode (JIT + RTGC), parce que le thread qui a provoqué la compilation n'est pas bloqué

Plan

- Rappels sur les systèmes temps réel
- Java et le temps réel : RTSJ
- Mise en œuvre : compilation, paramétrage
- **Annexe : JVM embarquées**

Les JVM embarquées

- Contraintes sur le matériel :
 - taille de la mémoire, vitesse du processeur, consommation
 - capacités graphiques réduites

- Contraintes logicielles :
 - ordonnancement, synchronisation
 - fonctionnement du GC
 - gestion des E/S, des interruptions
 - chargement dynamique des classes

JVM *Embarquées :KVM*

- Une JVM classique :
(512 Ko + système sous jacent + application) > 2Mo
- La KVM
 - taille : 128 Ko, dépend beaucoup des options sélectionnées :
 - floating point
 - taille du tas
 - cache byte code inhibé ou non
 - fonctionnement du GC
 - nombre de classes chargées
 - processeurs 16 bits (conçue pour les Palm, de 60 à 400Ko)
 - écrite en C ANSI, traduction souple bytecode en code natif
 - seule partie dépendante de la plateforme : allocation mémoire
 - **pb. un thread exécute N lignes d bytecode avant préemption (pas de vrai RR)**
- J2SE > J2EE > J2ME

J2ME

- J2ME est une version allégée de J2SE, adaptée aux appareils de faible puissance. Utilise deux types de spécifications, les configurations et les profils :
 - configurations
 - Connected Device Configuration (CDC), destinée aux du type PDA. Utilise une CVM, qui offre les mêmes fonctionnalités que la JVM classique
 - Connected Limited Device Configuration (CLDC), pour les téléphones portables, ... La machine virtuelle est la KVM qui ne possède pas toutes les fonctions de la JVM.
 - Les profils
 - Foundation pour CD (accès à toutes les fonctionnalités de J2SE)
 - MIDP pour CLDC (sous ensemble de J2SE, avec des classes d'entrée/sortie spécifiques, pb. de portage des IG)

L'API MIDP

- java.lang, un sous-ensemble de J2SE (pas de classe Float)
- java.io, pour obtenir des informations sur les systèmes distants.
- java.util contient seulement Calendar, Date, TimeZone, Enumeration, Vector, Stack, Hashtable et Random
- javax.microedition.io pour la classe Connector.
- javax.microedition.ui : l'interface utilisateur.
 - Pour respecter les contraintes d'adaptabilité au terminal, elle est structurée en deux niveaux :
 - haut niveau qui concerne l'aspect générique : Canvas, Graphics et Font (gestion écran, touches, ..),
 - bas niveau qui traite les spécificités du terminal
- javax.microedition.rms : le système de stockage persistant RMS (Record Management System) gère des " Record store" identifiés par leur recordID (openRecordStore, closeRecordStore, ...)

- javax.microedition.midlet contient MIDlet