Making Ideas a Reality


# Etat de l'art de Java temps réel

Ecole d'été Temps Réel –Nancy 2005

Marc richard-Foy

# Plan

- 1. Concepts et exemples de la concurrence Java
- 2. Critique du modèle de la concurrence Java
- 3. L'apport Java 5
- 4. La spécification Temps réel RTSJ
- 5. La spécification SCJ
- 6. Conclusion

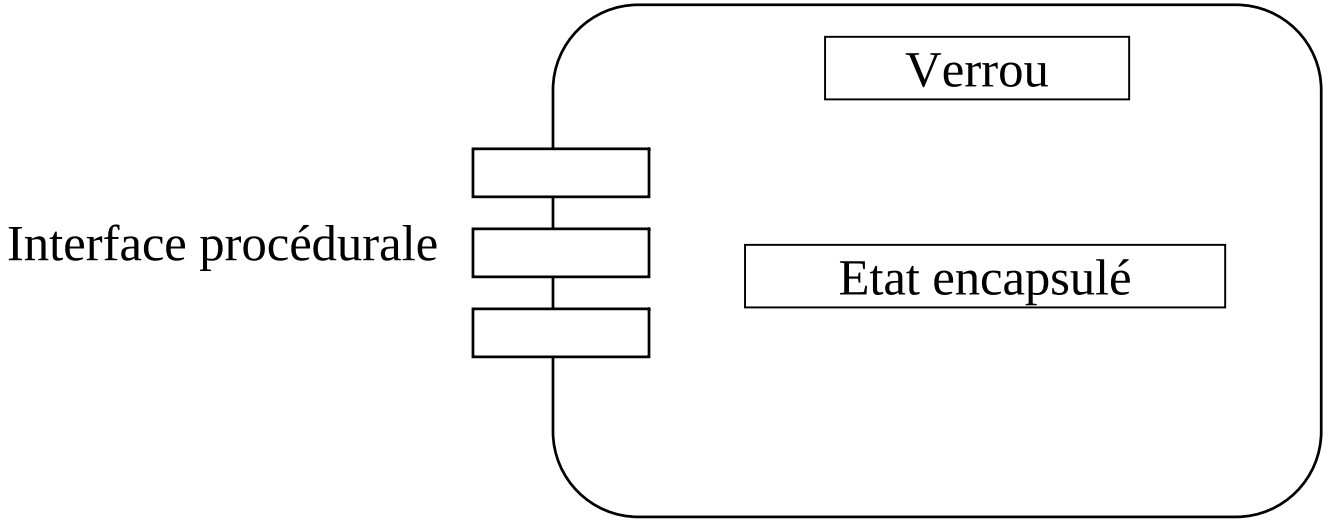
# Concepts de la concurrence Java (1)

- Il y a de nombreuses façons de supporter la programmation concurrente :
  - API pour la gestion explicite de processus concurrents
  - Mots clé dédiés dans le langage (ex: task en Ada)
  - Intégration dans la Programmation Orientée Objet avec par exemple la notion d'objets actifs
- Java adopte l'approche des objets actifs pour représenter les threads.
- Communication et synchronisation des threads
  - De nombreuses techniques : variables partagées, passage de messages ...
  - Java adopte la notion de moniteur aussi bien pour les mécanismes d'exclusion mutuelle que de communication asynchrone.

# Concepts de la concurrence Java (2)

- Un moniteur peut être considéré comme un objet où chacune de ses opérations s'exécute en exclusion mutuelle




- Les variables de conditions expriment une contrainte sur l'ordre d'exécution des opérations

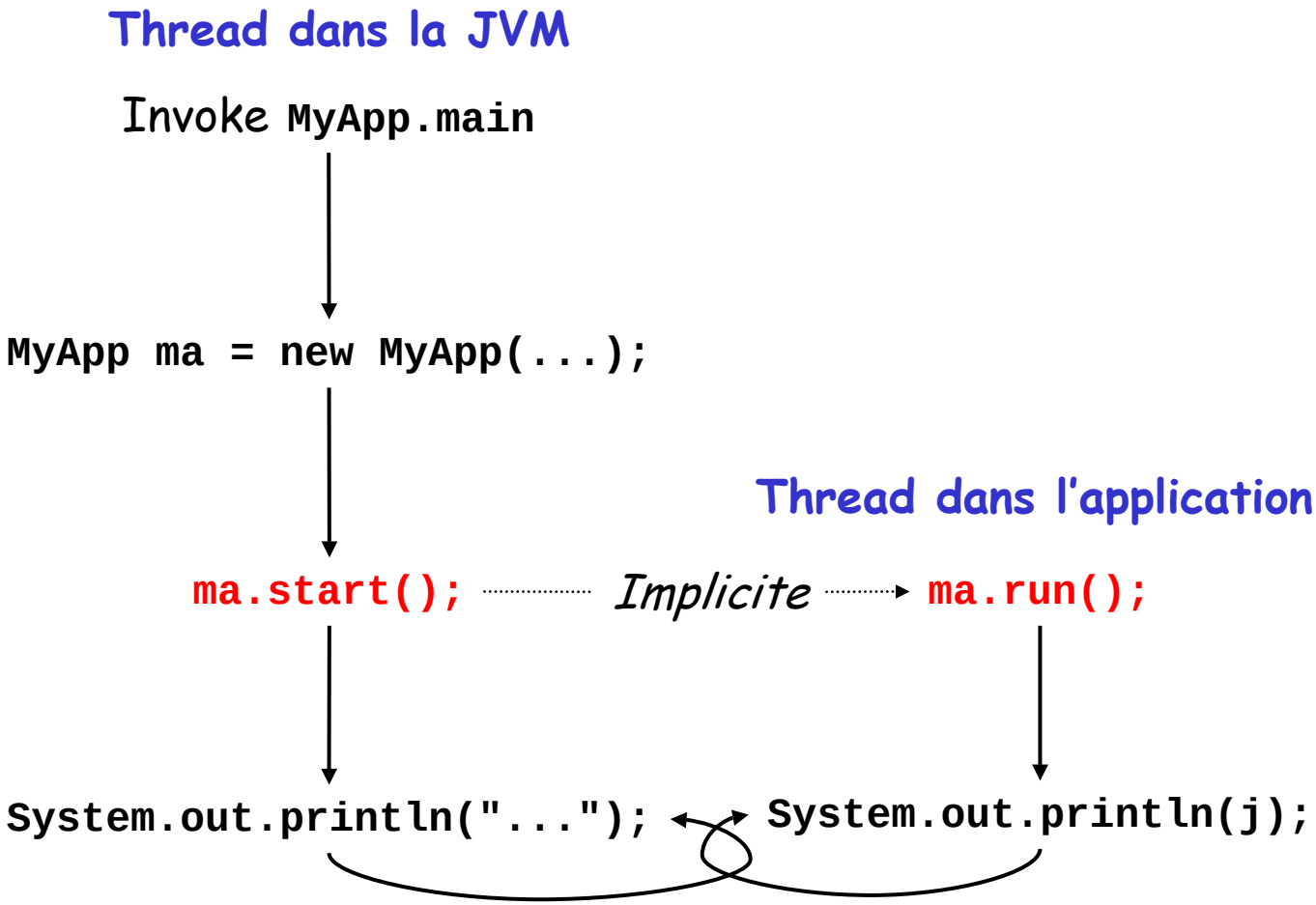

 

# Les threads – exemple (1)

```
/* Programme qui compte jusqu'à une valeur passée en
   argument de la ligne de commande. Chaque valeur de
   comptage est affichée à l'écran par le thread qui
   effectue le comptage tandis que le thread Main affiche
   un message indiquant qu'il a terminé. */

public class MyApp extends Thread{
    public final int count;
     public MyApp(int count){ this.count = count; }
    public void run(){
        for (int j=0; j<count; j++){
          System.out.println(j);
        }
    }
    public static void main( String[] args ){
        MyApp ma =
          new MyApp( Integer.parseInt(args[0]) );
        ma.start();
        System.out.println("Finished with main");
    }
}
```

# Les threads – exemple (2)





# Exclusion mutuelle - exemple

```
public class Coordinate {
  public Coordinate(int initX, int initY) {
    x = initX;
    y = initY;
  }
  public synchronized void update(int newX,
                                  int newY) {
    x = newX;
    y = newY;
  }
   public synchronized void display() {
    System.out.println(x);
    System.out.println(y);
  }
  private int x, y;
}
```

# Communication asynchrone - Exemple(1)

```
public class BoundedBuffer  {
  private int buffer[];
  private int first;
  private int last;
  private int count = 0;
  private int size;

  public BoundedBuffer(int length) {
    size = length;
    buffer = new int[size];
    last = 0;
    first = 0;
  };
```

# Communication asynchrone - Exemple(2)

```
    public synchronized void put(int item)
            throws InterruptedException {
        while (count == size) wait();
        last = (last + 1) % size ; // % is modulus
        count++;
        buffer[last] = item;
        notifyAll();
    };
    public synchronized int get()
            throws InterruptedException {
        while (count == 0) wait();
        first = (first + 1) % size ; // % is modulus
        count--;
        notifyAll();
        return buffer[first];
    };
}
```

# Plan

- 1. Concepts et exemples de la concurrence Java
- 2. Critique du modèle de la concurrence Java
- 3. L'apport Java 5
- 4. La spécification Temps réel RTSJ
- 5. La spécification SCJ
- 6. Conclusion

# Points forts du modèle de la concurrence Java

- Modèle simple et directement supporté par le langage
- Ceci limite potentiellement les erreurs qui surviennent en interfaçant les RTOS
- Le contrôle du typage fort et de la syntaxe du langage donne quelques protections
- Par exemple il n'est pas possible d'oublier la fin d'un bloc synchronized
- La portabilité des programmes est améliorée à cause du modèle de la concurrence indépendamment de l'OS sous-jacent

# Faiblesses (1)

- Mécanisme de priorité des threads insuffisant
- Programmation de processus périodiques limités par l'absence de mécanismes d'attente sur des temps absolus
- Le traitement des événements asynchrones nécessitent un thread dédié
- Transfert de contrôle asynchrone : Ne pas utiliser les méthodes stop(), suspend() et resume() sous risque d'interblocage ou d'incohérence

# Faiblesses (2) – wait/notify

- L'appel de moniteurs imbriqués peut conduire à des interblocages car le verrou du moniteur englobant n'est pas relâché quand le thread attend sur un wait du moniteur imbriqué.
- Seulement une file d'attente "wait set" par objet versus par condition associée à l'objet
- Origine du réveil d'un thread exécutant wait() : notify()/notifyAll(), interrupt() ou time-out ?
- Pas de préférence donnée aux threads qui continuent après un notify sur ceux qui accèdent au verrou du moniteur pour la première fois
- L'idiome "**`while (!condition) {obj.wait()}`**" est fondamental

# Faiblesses (2) -synchronized

Il n'est pas évident de déterminer quand des appels imbriqués à un moniteur peuvent être effectués :

- Les méthodes dans une classe n'étant pas déclarées **`synchronized`** peuvent néanmoins contenir des instructions **`synchronized`**;
- Les méthodes dans une classe n'étant pas déclarées **`synchronized`** peuvent être surchargées par une méthode **`synchronized`**;
- Une méthode non **`synchronized`** peut le devenir en étant appelée depuis une construction **`synchronized`** d'une sous-classe
- Les méthodes appelées via les interfaces ne peuvent pas être **`synchronized`**

# Faiblesses (3) – synchronized

- Impossible de faire machine arrière dans une tentative de prise de verrou, d'abandonner après un délai ou une interruption
- Impossible de personnaliser la sémantique du verrou en ce qui concerne la ré-entrance, l'équité ou les accès en lecture versus écriture.
- Pas de contrôle d'accès pour la synchronisation. Toute méthode peut effectuer synchronized(obj) pour tout objet accessible
- Impossibilité d'obtenir un verrou dans une méthode et de le relâcher dans une autre (structure en bloc)

# Plan

- 1. Concepts et exemples de la concurrence Java
- 2. Critique du modèle de la concurrence Java
- 3. L'apport Java 5
- 4. La spécification Temps réel RTSJ
- 5. La spécification SCJ
- 6. Conclusion

# Modèle Mémoire Java (1)

- Le modèle mémoire java (JMM) comprend:
  - Une mémoire principale partagées entre tous les threads
  - Une mémoire locale propre à chaque thread similaire à un cache
- Le modèle définit les conditions nécessaires et suffisantes qui permettent de savoir que les écritures mémoire des autres threads sont visibles par le thread courant et inversement que les écritures par le thread courant sont visibles par les autres threads.
- Les ré-ordonnancements des instructions par les compilateurs ou processeurs sont le souci principal
- http://www.cs.umd.edu/users/pugh/java/memoryModel

# Modèle Mémoire Java (2)

- Java 1.5 a révisé la sémantique des mots clé **`final`**, **`volatile`** et **`synchronized`** pour assurer qu'un programme correctement synchronisé s'exécute correctement sur tout type d'architecture matérielle

- La synchronisation représente plus qu'une simple exclusion mutuelle :
  - En sortie d'un bloc "synchronized" le cache du thread courant est sauvé dans la mémoire partagée
  - A l'entrée d'un bloc "synchronized" les variables locales du thread courant sont rechargées depuis la mémoire partagée

# Exemple avec volatile

```
class VolatileExample {
   int x= 0;
   volatile boolean v = false;
   public void writer() {
      x=42;
      v= true;
   }
   public void reader() {
      if (v == true) {
      // si on utilise x on a la garantie de voir 42
      {
   {
}
```

Ce qui est visible par le thread A quand il écrit le champ volatile le devient par le thread B quand il lit ce même champ !

# Mécanisme étendu de verrou

```
package java.util.concurrent.locks;
public interface Lock {
  public void lock();
    // Attendre d'obtenir le verrou.
  public Condition newCondition();
    // Créer une variable de condition
    // à utiliser avec le verrou.
  public void unlock();
  …
}
```

# Les variables de condition

```
package java.util.concurrent.locks;
public interface Condition {
  public void await()
          throws InterruptedException;
    // Relâche atomiquement le verrou associé
    // et met le thread courant en attente.
  public void signal();
   // Réveil un thread en attente du verrou.
  public void signalAll();
   // Réveil tous les threads en attente
   // du verrou.
  …
}
```


# Utilitaires de gestion de la concurrence (1)

- Le paquetage **`java.util.concurrent`** fournit un ensemble très riche et complet de classes et interfaces pour le développement d'applications concurrentes
- Locks (**`Lock`**, **`ReadWriteLock`** et **`Condition`**)
  - time-out possible,
  - interrupt possible,
  - conditions multiples par verrou,
  - pas la structure en région,
  - personalité de l'accès au verrou (read/write),
  - équité possible dans le mécanisme d'attente,
  - respecte le modèle mémoire Java
- Atomicité **`java.util.concurrent.atomic`**

# Utilitaires de gestion de la concurrence (2)

- "Thread-safe" collections
  - Interfaces **`Queue`** et **`BlockingQueue`**
- Task Scheduling Framework
  - Séquencer et contrôler des tâches asynchrones
  - Soumission dans un thread ou un pool de thread
  - Interfaces **`Callable`**, **`Future`** et **`Executor`**
- Synchronizers
  - **`Semaphore`** et **`Mutex`**
  - **`CyclicBarrier`**
  - **`CountDownLatch`** (compte à rebours)
  - **`Exchanger`** (rendezvous)
- Résolution du temps à la nanoseconde pour toutes les méthodes avec timeout

# Classe LockSupport

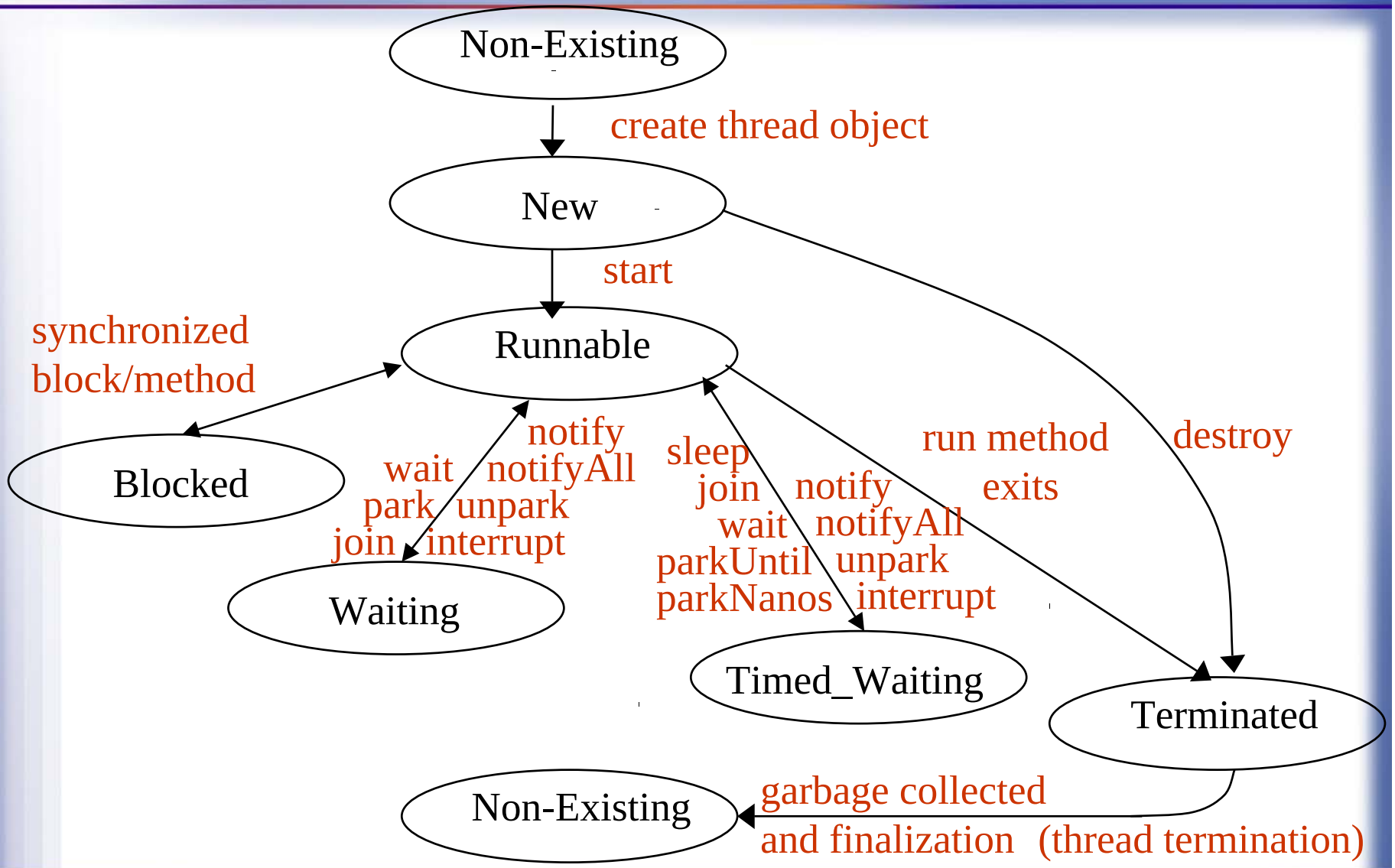
- La classe **`LockSupport`** fournit les mécanismes de base pour le support des verrous :

```
package java.util.concurrent.locks;
public class LockSupport {
   public static void park()
   public static void parkNanos(long nanos)
   public static void parkUntil(long deadline)
   public static void unpark(Thread thread)
}
```

- Semblable au sémaphore binaire
- Utiliser **`park/unpark`** au lieu des méthodes déconseillées **`Thread.suspend`** et **`Thread.resume`**
- Briques de base pour les implementeurs qui souhaitent développer leurs propres classes Locks.

# Etat des threads Java





# Plan

- 1. Concepts et exemples de la concurrence Java
- 2. Critique du modèle de la concurrence Java
- 3. L'apport Java 5
- 4. La spécification Temps réel RTSJ
- 5. La spécification SCJ
- 6. Conclusion


# RTSJ principes directeurs

- Compatibilité arrière avec les programmes java non temps réel
- Support du principe “Write Once, Run Anywhere” mais pas au détriment du déterminisme
- Supporter l’état de l’art actuel tout en étant ouvert à des évolutions
- Donner la priorité au déterminisme dans la conception
- Pas d’extension syntaxique du langage
- Donner des degrés de liberté aux implémentations

# Améliorations temps réel

- RTSJ améliore la programmation temps réel Java dans les domaines suivants :
  - Gestion mémoire
  - Horloge et gestion du temps
  - Ordonnancement et objets “schedulable”
  - Threads temps réel
  - Gestion des événemenst asynchrones et timers
  - Trasnfert de contrôle asynchrone
  - Synchronisation et partage de ressource
  - Accès à la mémoire physique

# Modèle mémoire RTSJ

- **`HeapMemory`** : c'est le tas mémoire Java classique.
- **`ImmortalMemory`** : c'est une zone mémoire partagée entre toutes les threads. Les objets alloués dans cette zone ne sont jamais collectés par le Garbage Collector et ne sont libérés que lorsque le programme se termine.
- **`ScopedMemory`** : c'est une zone pour les objets qui ont une durée de vie bien définie. Un compteur de références associé à chaque ScopedMemory permet de conserver la trace de combien d'entités temps réel sont à un moment donné en train d'utiliser cette région mémoire.

# ScopedMemory - exemple

```
import javax.realtime.*;
public class Region {
  LTMemory locale;
  class Action implements Runnable {
    public void run() {
    // Tous les objets alloués dans cette
    // méthode run() ont la portée de
    // l'exécution de cette méthode : ils
    // seront libérés en sortant de la méthode.
     ... }
  }
  static public void main(String [] args){
    locale = new LTMemory(8*1024, 8*1024);
    Action action = new Action();
    // Pendant l'exécution de la méthode run()
    // associée à l'argument action  les objets
    // seront alloués dans la région locale.
    locale.enter(action);
  }
}
```

# Gestion du temps

- La classe abstraite **`HighResolutionTime`** définit le temps avec une précision de la nanoseconde
- Les classes **`AbsoluteTime`**, **`RelativeTime`** et **`RationalTime`** permettent de manipuler des temps absolus (dates) ou des intervalles de temps (durées)
- Ces temps sont relatifs à des horloges. Plusieurs types d'horloge sont possibles au sein d'un même programme
- RTSJ fournit toujours une horloge temps réel associé à un temps à croissance monotone

# Objets Schedulable

- RTSJ généralise les entités “schedulable” du thread jusqu’à la notion d’objets **`Schedulable`**
- Un objet **`Schedulable`** implémente l’interface **`Schedulable`**
- Chaque objet **`Schedulable`** indique ses exigences:
  - D’échéance (quand il doit être élligible),
  - Mémoire (ex: taux d’allocation sur la heap),
  - Ordonnancement (priorité)
- L’interface **`Schedulable`** est implémentée par les classes:
  - **`RealTimeThread`** et **`NoHeapRealTimeThread`** qui étendent la classe Thread pour le temps réel
  - **`AsyncEventHandler`** pour le traitement d’événements asynchrones (handlers d’interruptions par exemple)

# Thread périodiques

```
public class Periodic extends RealtimeThread{
  public Periodic(
    PriorityParameters PP, PeriodicParameters P)
  { super(pp, p); };

  public void run(){
   boolean noProblems = true;
   while(noProblems) {
     // Code à exécuter à chaque période
     ...
     noProblems = waitForNextPeriod();
   }
   // Echéance manquée
   ...
  }
}
```

# Priority Inversion control in RTSJ

- L'inversion de priorité peut survenir quand un objet schedulable est bloqué en attente de ressource
- Pour limiter la durée du temps de blocage RTSJ impose que :
  - Toutes les files d'attente du système soient gérées par ordre de priorité
  - FIFO pour les objets schedulable de même priorité dans la même file
  - De même les files associées à l'appel de la méthode wait() de la classe Object doivent être aussi ordonnées par priorité
  - Par défaut le protocole d'héritage de priorité est appliqué aux objets schedulable bloqués en attente de ressource
  - Le protocole d'héritage à priorité plafond (PCP) peut être associé à une ressource de façon programmatique

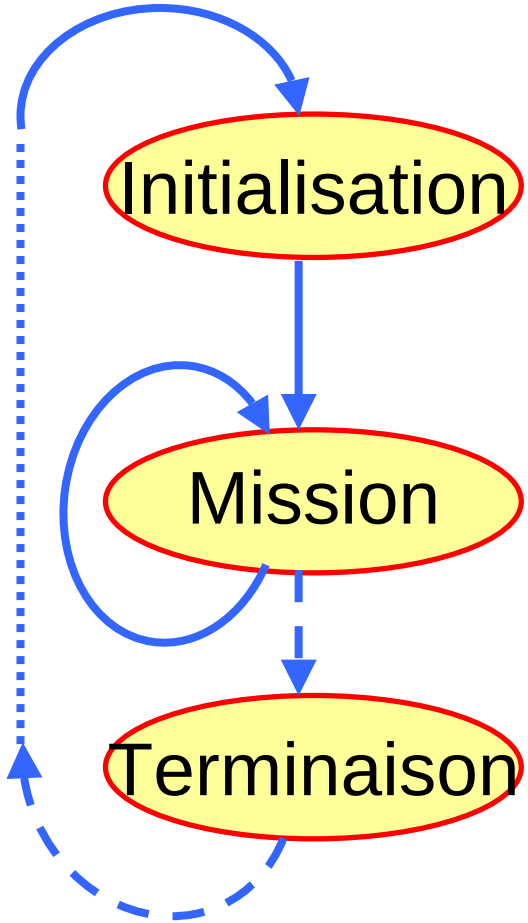

 

# Plan

- 1. Concepts et exemples de la concurrence Java
- 2. Critique du modèle de la concurrence Java
- 3. L'apport Java 5
- 4. La spécification Temps réel RTSJ
- 5. La spécification SCJ
- 6. Conclusion

# Systèmes critiques - Modes d'exécution




Mise sous tension
Démarrage

Mode Opérationnel

Récupération
Redémarrage


# Modèle de concurrence SCJ

- L'ordonnancement est préemptif par priorité fixe,
- Seuls les objets **`Schedulable`** globaux sont supportés,
- **`ReleaseParameter`** soit périodiques soit sporadiques,
- Chaque traitement d'événement asynchrone s'exécute dans un thread dédié,
- Les dépassements d'échéance sont détectés,
- Le partage de ressource se fait à l'aide de méthodes synchronized (non blocantes),
- L'inversion de priorité est contrôlée par le PCP

# Plan

- 1. Concepts de la concurrence Java
- 2. Exemples illustrant le modèle de la concurrence Java
- 3. Critique du modèle de la concurrence Java
- 4. L'apport Java 5 : le JSR 133
- 5. L'apport Java 5 : le JSR 166
- 6. La spécification Temps réel RTSJ
- 7. La spécification SCJ
- 8. Conclusion

# Conclusion

- Java 1.5 augmente significativement les capacités de la programmation concurrente,
- Le RTSJ version 1.0.1 est stable,
- Le SCJ a été soumis comme JSR
  - The Open Group est le chef de file
  - Discussions en cours sur le type d'annotations pour l'analyse temps réel et le modèle mémoire (régions imbriquées)
- Une voie intéressante : les systèmes temps réel à architecture neutre