Le langage de programmation **JR**

IFT630
Gabriel Girard

27 janvier 2021

Table des matières

1 Introduction

JR est un langage de programmation qui permet de faire des programmes séquentiels, parallèles et répartis. C'est une extension de Java.

2 Survol du langage

JR a été créé spécialement pour résoudre des problèmes concurrents. JR ajoute par-dessus Java des éléments de programmation qui facilitent l'utilisation de fonctionnalités déjà présentes dans Java et ajoute aussi au langage des fonctionnalités directement reliées au parallélisme. En particulier, il ajoute des éléments qui facilitent la création de processus et l'utilisation de différents outils de synchronisation ou de communication, tels les sémaphores, les moniteurs et le passage de messages.

Le fait que JR intègre plusieurs facilités de programmation et de synchronisation dans le même langage le rend particulièrement intéressant pour l'enseignement de la programmation parallèle.

JR est une implantation en JAVA du langage de programmation parallèle SR. Les programmes sources de JR sont traduits en Java et exécutés sur la machine virtuelle Java comme toutes les autres classes Java.

Dans ce document, je suppose que vous connaissez déjà le langage Java. Je présente seulement les concepts de programmation parallèle introduits par JR.

3 Compilation et exécution d'un programme JR

Pour introduire la compilation et l'exécution d'un programme JR, nous commençons par un programme qui imprime la chaîne "Bonjour le monde".

Il suffit donc de créer un fichier intitulé «`Bonjour.jr`» contenant le code donné dans le programme 1 (du Java pur).

Programme 1 : Premier exemple de programme JR

```
public class Bonjour
{
    public static void main(String[] args)
    {
        System.out.println("Bonjour le monde");
    }
}
```

Il existe deux façons de compiler/exécuter un programme JR :

1. Compilation et exécution séparée
   - jrc Bonjour.jr
   - jrrun Bonjour
2. Compilation et exécution combinées
   - jr Bonjour

     Note : il est important d'omettre l'extension ".jr".

4 Ajouts important

Avant de parler des ajouts à JR permettant de faire de la concurrence, de la synchronisation et de la communication, il faut introduire quelques concepts de JR qui facilitent l'implantation de ces ajouts. Ces concepts sont fortement inspirés du langage de programmation parallèle SR.

4.1 Opérations

Une opération est une méthode déclarée avec le mot clé «`op`». En JR, une opération est plus versatile qu'une simple méthode car elle peut être appelée de différentes façons comme nous le verrons plus tard.

4.1.1 Déclaration des opérations

Une «opération» en JR (ou méthode `op`) prend la même forme qu'une méthode normale mais utilise le mot clé `op`. Une opération peut être appelée de la même façon qu'une méthode normale ou avec les énoncés `call` et `send`. Le programme 2 montre la syntaxe pour la déclaration d'une opération.

Programme 2 : Syntaxe de la création d'un opération

```
public static op void fin()
{
    System.out.println("C'est la fin")
}
```

Une opération peut être appelée comme toutes les autres méthodes. Les programmes 3 et 4 montrent comment utiliser les opérations.

Programme 3 : Exemple d'utilisation des «op»

```
public class Exemple1
{
    public static op int addition(int i1, int i2)
    {
        System.out.println("addition()");
        return i1 + i2;
    }
    public static void main(String... args)
    {
        System.out.println(addition(2, 2));
        call addition(3, 3);
    }
}
```

Programme 4 : Exemple d'utilisation des «op»

```
public class Exemple2
{
    private static op int carre(int x)
    {
        System.out.println("Dans carre " + x);
        return x * x;
    }
    public static void main(String... args)
    {
        System.out.println(carre(23));
        carre(41);
        call carre(41);
    }
}
```

4.1.2 Déclaration séparée des opérations

La déclaration d'une opération peut aussi se séparer en deux parties. On peut l'appeler la déclaration longue. Cela permet de définir une opération au même titre que la déclaration «courte». Toutefois la déclaration longue possède plusieurs avantages que nous verrons plus tard. En particulier, il est aussi possible de déclarer des vecteurs d'opération. Les programmes 5 et 6 donnent des exemples de déclarations.

Programme 5 : Exemple d'utilisation des «op» en JR

```
public static op int addition(int i1, int i2);

public static int addition(int i1, int i2)
{
    System.out.println("addition()");
    return i1 + i2;
}
```

4.1.3 Appels aux méthodes

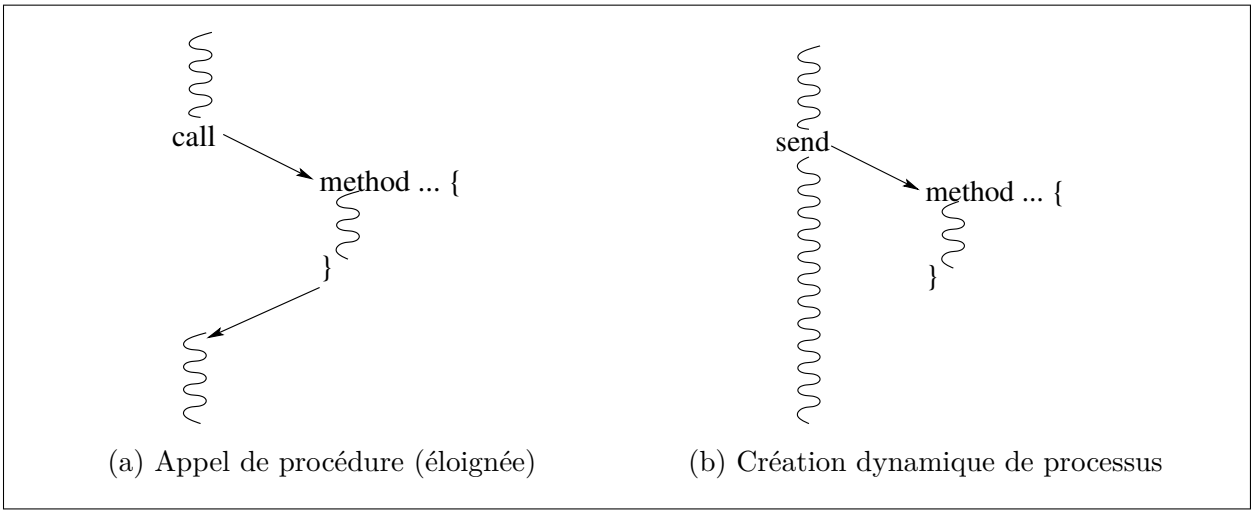
JR permet aussi d'appeler une opération grâce à l'énoncé «send». Dans ce cas, le «send» agit comme un «fork» et démarre un nouveau processus qui exécute l'opération. L'appelant

Programme 6 : Exemple d'utilisation des «op» en JR

```
private static op int carre(int x);

private static int carre(int x)
{
    System.out.println("Dans carre " + x);
    return x * x;
}
```

et l'appelé s'exécutent alors en parallèle. La figure 1 montre l'effet du «call» et du «send».



(a) Appel de procédure (éloignée) (b) Création dynamique de processus

FIGURE 1 – Appel de méthode avec «call» et «send»

4.2 Capacités

Les capacités sont une forme de pointeurs de fonctions. Elles servent donc à référer aux opérations. Le Programme 7 montre comment déclarer une capacité.

Programme 7 : Syntaxe pour la création d'une capacité

```
cap signature_de_opération identificateur_capacité
```

La capacité s'utilise de la même façon que l'opération à laquelle elle réfère. Elle prend les mêmes paramètres et retourne une donnée du même type. La «signature de l'opération» est formée du type retourné par l'opération et de ses paramètres en omettant le nom de la méthode. Une fois la capacité déclarée, il est possible de lui assigner toute opération ayant la signature spécifiée.

On utilise la capacité de la même façon qu'une méthode normale. Elle peut aussi être passée en paramètre. Les programmes 8, 9 et 10 montrent comment utiliser les capacités.

Programme 8 : Exemple de «capacité» JR

```
import java.util.Random;
public class Capacité1
{
    public static op int addition(int i1, int i2)
    {
        System.out.println("addition()");
        return i1 + i2;
    }
    public static void main(String... args)
    {
        cap int (int, int) cap_addition; // Déclaration de la capacité
        cap_addition = addition;         // Référence vers l'opération
        System.out.println(cap_addition(2, 2));
    }
}
```

Programme 9 : Exemple de «capacité» JR

```
public class Capacité2
{
    // déclarations des opérations
    public static op void d(int);
    public static op int e(int);
    public static op double f(double);
    public static op double g(double);
    // Déclarations des capacités
    private static cad void (int) x, z;
    private static cad double (double) y;

    public static void main(String [] args)
    {
        x = d;  // x pointe cers l'opération d
        x(387); // appel à l'opération d

        // faire pointer y vers f ou g
        if (e(9) > 0) {y = f;}
        else          {y = g;}
        // Appel à la fct pointé par y
        System.out.println( y(4.351));
        // Les capacités peuvent être assignées ou comparées
        z = x;
        if (y == f) {System.out.println("y is f");}
        else        {System.out.println("y is g");}
    }
}
```

Programme 10 : Exemple de «capacité» JR

```
import java.util.Random;
public class Capacité3
{
  public static op int carre(int x)  {  return x * x; }
  public static op int dec(int x)     {  return x - 3; }
  public static op int inc(int x)     {  return x + 11; }

  public static op int add(cap int (int) f1, cap int (int) f2, int result)
  {   return f1(result) + f2(result); }
  public static op int sub(cap int (int) f1, cap int (int) f2, int result)
  {   return f1(result) + f2(result); }

  public static void main(String... args)
  {
    cap int (int) fcts[] = new cap int (int)[3];
    fcts[0] = carre;
    fcts[1] = dec;
    fcts[2] = inc;
    cap int (cap int (int), cap int (int), int)[] double_fonctions
              = new cap int (cap int (int), cap int (int), int)[2]
    double_fcts[0] = add;
    double_fcts[1] = sub;
    int result = 100;
    Random random = new Random();

    for(int i = 0; i <= 10; i++)
    {
       int idx1 = random.nextInt(double_fcts.length);
       int idx2 = random.nextInt(fcts.length);
       res = double_fcts[index1](fcts[idx2], fcts[idx2], res);
    }
    System.out.println(res);
  }
}
```

5 Processus

Le principal intérêt de JR est la facilité avec laquelle il est possible de créer des processus. En JR, contrairement à Java, il n'est pas nécessaire d'instancier des objets. Il suffit de déclarer des processus.

5.1 Survol rapide

Pour déclarer des processus, JR introduit le mot réservé `process`. Le programme 11 montre un déclaration simple de processus.

Programme 11 : Syntaxe pour la création de processus

```
process Pcs1
{
    System.out.println("Processus");
}
```

La déclaration est simple et il n'est pas nécessaire de faire une démarrage explicite. Il suffit de créer une instance de la classe pour que le processus démarre. Un processus peut aussi être déclaré `static`. Dans ce cas, il ne sera pas démarré lors de l'instanciation de la classe mais lors de la résolution de la classe par la machine virtuelle.

Dans le programme 12, on démarre un processus qui affiche à partir d'un fil d'exécution. Le programme 13 montre un exemple de calcul utilisant deux processus.

Programme 12 : Exemple de création de processus

```
public class ProcessusBonjour
{
    static process bonjour
    {
        System.out.println("Bonjour le monde");
    }
    public static void main(String[] args){}
}
```

Programme 13 : Exemple avec deux processus

```
public class DeuxPcsSimple
{
    private static int x = 0;
    private static process p1  {    x += 3;   }
    private static process p2  {    x += 4;   }
    public static void main(String[] args){}
}
```

JR permet de déclarer un ensemble de fils d'exécution en un seul énoncé avec une syntaxe qui ressemble à celle d'une itération. Le programme 14 donne la syntaxe pour le démarrage

de «n» fils d'exécution. Le programme 15 déclare et démarre 25 fils qui affichent `bonjour`. Le résultat de l'exécution du programme 15 est donné à la figure2.

Programme 14 : Syntaxe pour la création de processus en JR

```
static process Hello((int id = 0; id <= n; id++)){}
```

Programme 15 : Exemple de création de processus en JR

```
public class ProcessusBonjour25
{
    static process bonjour((int id = 0; id <= 25; id++))
    {
        System.out.println("Vous avez le bonjour du fil  no. " + id);
    }
    public static void main(String[] args){}
}
```

```
Vous avez le bonjour du fil  no. 24
Vous avez le bonjour du fil  no. 5
Vous avez le bonjour du fil  no. 4
Vous avez le bonjour du fil  no. 25
Vous avez le bonjour du fil  no. 3
Vous avez le bonjour du fil  no. 2
Vous avez le bonjour du fil  no. 23
Vous avez le bonjour du fil  no. 1
Vous avez le bonjour du fil  no. 0
Vous avez le bonjour du fil  no. 18
Vous avez le bonjour du fil  no. 20
Vous avez le bonjour du fil  no. 16
Vous avez le bonjour du fil  no. 17
Vous avez le bonjour du fil  no. 15
Vous avez le bonjour du fil  no. 12
Vous avez le bonjour du fil  no. 14
Vous avez le bonjour du fil  no. 19
Vous avez le bonjour du fil  no. 22
Vous avez le bonjour du fil  no. 21
Vous avez le bonjour du fil  no. 13
Vous avez le bonjour du fil  no. 11
Vous avez le bonjour du fil  no. 10
Vous avez le bonjour du fil  no. 9
Vous avez le bonjour du fil  no. 8
Vous avez le bonjour du fil  no. 7
Vous avez le bonjour du fil  no. 6
```

FIGURE 2 – Résultat de l'exécution du programme 15

5.2 Syntaxe complète

La syntaxe exacte pour la création de processus est donnée au programme 16. Il est important de noter qu'il y a une paire de parenthèses qui englobent l'ensemble des quantificateurs et d'autre paires de parenthèses qui englobent que chaque quantificateur individuellement. Le format d'un quantificateur, similaire au format du `for`. Il est présenté dans le programme 17.

Le porgramme 18 montre un exemple de création de processus sans `static`. Les processus sont créés seulement lorsque le `new` est exécuté. Chaque instanciation de l'objet crée une nouvelle variable `x` et deux nouveaux processus.

Le programme 19 déclare et démarre 100 fils qui affichent `bonjour`. Le programme 20 démarre 4 processus. L'expression «`i != 0`» est la condition «`such that`» qui ajoute une condition sur la création de processus.

Programme 16 : Syntaxe de la création d'un opération

```
   process id_pcs { énoncés }
ou
   process id_pcs ( (quantificateur), (quantificateur), ... ) { énoncés }
```

Programme 17 : Format des quantificateurs

```
   (expr. initiale ; continuation ; incrément)
ou
   (expr. initiale ; continuation ; incrément ; expression "such that")
```

Programme 18 : Exemple de déclaration de processus non `static`

```
public class pcs1
{
  public static void main(String [] args)
  {
      new Obj_pcs();
  }
}
public class Obj_pcs
{
    private int x = 0;
    private static process p1
    {
      x += 3;
      system.out.println("Pcs 1");
    }
    private static process p2
    {
        x += 4;
        system.out.println("Pcs 1");
    }
}
```

Programme 19 : Exemple de création de processus en JR

```
import edu.ucdavis.jr.JR;

public class pcs5
{
  static process bonjour((int id1 = 0; id1 < 10; id1++),
                         (int id2 = 0; id2 < 10; id2++))
  {
     System.out.println("Vous avez le bonjour du fil  no. " + id1 + id2);
  }
  public static void main(String[] args){}
}
```

Programme 20 : Exemple de création de processus en JR

```
import edu.ucdavis.jr.JR;

public class pcs5a
{
    static process bonjour((int id1 = 0; id1 < 10; id1++; id!=0))
    {
        System.out.println("Vous avez le bonjour du fil no. " + id1 );
    }
    public static void main(String[] args){}
}
```

5.2.1 Création dynamique de processus

La syntaxe précédente est utile pour créer des processus quand le nombre de processus à créer est connu à l'avance. Dans certains cas, on doit pouvoir créer dynamiquement des processus pendant l'exécution. Pour ce faire, on utilise l'instruction `send` sur une opération. Les programmes 21 à 25 donnent des exemples d'utilisation du `send` pour démarrer des processus.

Programme 21 : Exemple de création dynamique de processus

```
public class pcs6
{
    private static int x = 0;
    private static op void p1();
    static { send p1(); }
    private static op void p2();
    static { send p2(); }
    private static void p1()  { x += 3; System.out.println("p1"); }
    private static void p2()  { x += 4; System.out.println("p2"); }
    public static void main(String[] args){}
}
```

Programme 22 : Exemple de création dynamique de processus

```
public class pcs7
{
    private static int x = 0;

    private static op void p1();
    private static op void p2();
    public pcs7()  // constructeur
    {
        send p1();
        send p2();
    }
    private static void p1()  { x += 3; System.out.println("p1"); }
    private static void p2()  { x += 4; System.out.println("p2"); }
    public static void main(String[] args){}
}
```

5.2.2 Création de processus statiques et non-statiques

La distinction entre des processus statiques ou non, est la même que pour les autres classes ou objets. Le programme 26 donne un exemple de création d'un pro=cessus statique et d'un processus non statique. Le programme 27 montre une mauvaise utilisation des processus statiques. Comme les processus statiques sont démarrés avant toute exécution du code de la classe, la variable N ne sera pas initialisée avant la création des processus. Il n'y aura donc aucune création de processus.

Programme 23 : Exemple de création dynamique de processus

```
public class pcs8
{
    private static op void p(int);
    static
    {  for(int i = 0; i < 8; i++) { send p(i); }  }
    private static void p(int i)
    {  System.out.println("Pcs p" + i); }
    private static op void q(int);
    static
    {
        for(int i = 0; i <= 5; i++)
        { if (i != 3) send q(i); }
    }
    private static void q(int i)
    {
        System.out.println("Pcs q" + i);
    }
    public static void main(String[] args){}
}
```

Programme 24 : Exemple de création dynamique de processus

```
import edu.ucdavis.jr.JR;

public class pcs9
{
  public static class Compresseur
  {
     public op void compresse(String);

     public void compresse (String nomFichier)
     {
        // On fait semblant de compresser...
        System.out.println("Début compression... " + nomFichier);
        JR.nap(10);
        System.out.println("Fin compression... " + nomFichier + " ...");
     }
  }
  public static void main(String[] args)
  {
      Compresseur c = new Compresseur();
      send c.compresse("Toto");
      send c.compresse("TATA");
  }
}
```

Programme 25 : Exemple de création dynamique de processus

```
import edu.ucdavis.jr.JR;

public class pcs10
{
    public static process p((int id =1; id <= 2; id++))
    {
        System.out.println("Vous avez le bonjour du fil  no. " + id);
        JR.nap(10);
        System.out.println("ADIEU du fil  no. " + id);
    }

    public static void main(String[] args)
    { send p(17);}
}
```

Programme 26 : Exemple de création de processus statiques et non statiques

```
import edu.ucdavis.jr.JR;
public class pcs11
{
   // Processus serveur statique créé au démarrage du programme
   public static process serveur
   {
      System.out.println("Vous avez le bonjour du serveur. ");
      JR.nap(10);
      System.out.println("ADIEU du serveur ");
    }

    // processus client créé lors de la création d'une instance de l'objet
    private process client
    {
      System.out.println("Vous avez le bonjour du client. " + id);
      JR.nap(10);
      System.out.println("ADIEU du client " + id);
    }
    private int id;

    pcs11(int id)
    { this.id = id;}

    public static void main(String[] args)
    {
       // création de 2 instances de l'objet et de 2 processus clients
       new pcs11(1);
       new pcs11(2);
    }
}
```

Programme 27 : Programme sans création de processus

```
import edu.ucdavis.jr.JR;

public class pcs12
{
    private static N;
    public static process p((int i = 0; i < N; i++))
    {
        System.out.println("Vous avez le bonjour de p " + i);
    }
    public static void main(String[] args)
    {
        N = Integer.parseInt(args[0]);
    }
}
```

6 Action finale

JR introduit un nouveau concept appelé «action finale». C'est une action qui s'exécute lorsque le système ne fait plus rien. Cela peut se produire quand tous les processus sont terminés ou en interblocage.

Pour détecter et agir lorsque le système termine, on utilise une opération comme action finale. On peut la déclarer comme l'action à exécuter quand le système ne fait plus rien.

Le programme 28 montre comment définir et utiliser une action finale. Dans cet exemple, on utilise la méthode `registerQuiescenceAction(op)` de la classe JR qui fournit quelques méthodes utilitaires pour les programmes JR. Le résultat de l'exécution est donné à la figure 3.

Programme 28 : Exemple d'action finale

```
import edu.ucdavis.jr.JR;

public class ActionFinale
{
    static process Bonjour((int id = 0; id <= 25; id++))
    {
        System.out.println("Bonjour a tous du fil no. " + id);
    }
    public static void main(String[] args)
    {
        try
        {  JR.registerQuiescenceAction(fin);    }
        catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
        {  e.printStackTrace();  }
    }
    public static op void fin()
    {
        System.out.println("C'est la fin.");
    }
```

Ce genre d'action peut être utile pour afficher un résultat final suite à des calculs menés par plusieurs fils et plus particulièrement pour les situations où il y a interblocage. Le programme 29 montre l'utilisation d'une action finale pour afficher un résultat. Le résultat de l'exécution de ce programme suivant produit des résultats variables qui sont illustrés à la figure 4.

7 Synchronisation

7.1 Sémaphores

L'utilisation des sémaphores en JR est relativement simple. Il suffit de déclarer un sémaphore avec une valeur initiale et de l'utiliser avec les énoncés `P` et `V`. Le programme 30

```
Bonjour a tous du fil no. 0
Bonjour a tous du fil no. 22
Bonjour a tous du fil no. 23
Bonjour a tous du fil no. 21
Bonjour a tous du fil no. 24
Bonjour a tous du fil no. 20
Bonjour a tous du fil no. 19
Bonjour a tous du fil no. 18
Bonjour a tous du fil no. 17
Bonjour a tous du fil no. 16
Bonjour a tous du fil no. 15
Bonjour a tous du fil no. 14
Bonjour a tous du fil no. 13
Bonjour a tous du fil no. 12
Bonjour a tous du fil no. 11
Bonjour a tous du fil no. 10
Bonjour a tous du fil no. 9
Bonjour a tous du fil no. 8
Bonjour a tous du fil no. 7
Bonjour a tous du fil no. 6
Bonjour a tous du fil no. 5
Bonjour a tous du fil no. 4
Bonjour a tous du fil no. 3
Bonjour a tous du fil no. 2
Bonjour a tous du fil no. 1
C'est la fin.
```

FIGURE 3 – Résultat de l'exécution d'une action finale.

Programme 29 : Exemple d'action finale

```
import edu.ucdavis.jr.JR;
public class DeuxPcs
{
    private static int x = 0;
    private static process p1
    {
        x += 3;   System.out.println("Dans p1");
    }
    private static process p2
    {
        x += 4;   System.out.println("Dans p2");
    }
    public static void main(String[] args)
    {
        try
        {   JR.registerQuiescenceAction(end); }
        catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
        {   e.printStackTrace();  }
    }
    public static op void end()
    {
        System.out.printf("C'est la fin .. ");
        System.out.printf(" x = %d\n ", x);
    }
}
```

```
  Exécution 1             |       Exécution 2         |       Exécution 3
__________________________|___________________________|_____________________________________
  Dans p1                 |  Dans p2                  |  Dans p2
  Dans p2                 |  Dans p1                  |  Dans p1
  C'est la fin ..  x = 7  |  C'est la fin ..  x = 3   |  C'est la fin ..  x = 4
```

FIGURE 4 – Résultat de l'exécution d'une action finale.

montre comment utiliser les sémaphores. Le programme 31 donne un exemple complet d'utilisation d'un sémaphore pour l'exclusion mutuelle. Si un sémaphore est initialisé à une valeur différente de 1, on peut faire de la synchronisation conditionnelle ou du contrôle d'accès.

Le problème avec cette solution ou toute solution utilisant les sémaphores est la performance. En effet, si on fait 100 itérations avec 100 fils d'exécution, l'exécution avec les sémaphores sera très lente. Le programme 32 n'utilise pas les sémaphores et s'exécute beaucoup plus rapidement que la version avec sémaphores. Malheureusement il ne donne pas toujours la bonne réponse. L'utilisation des variables de type «mutex» en Java, C++ ou autres langages peut améliorer la performance grâce à une implantation plus efficace du sémaphore (qui est binaire dans ce cas).

Programme 30 : Syntaxe pour la création et l'utilisation d'un sémaphore

```
sem mutex = 1;
...
P(mutex);
//Critical section
V(mutex);
```

Programme 31 : Exemple d'utilisation d'un sémaphore en JR

```
private static sem mutex  = 1;
private static int valeur = 0;
static process Calculatrice((int id = 0; id < 50; id++))
{
    for(int i = 0; i < 5; i++)
    {
        P(mutex);
        valeur = valeur + 2;
        V(mutex);
    }
}
```

7.1.1 Vecteur de sémaphores

Il est possible d'avoir un tableau de sémaphores en JR. Pour cela il faut définir un tableau de capacités et chaque élément du tableau doit être explicitement défini. Les programmes 33 et 34 montrent comment déclarer un tableau de sémaphores.

Le programme 35 utilise un tableau de sémaphores pour que les processus se passent le contrôle à tour de rôle dans un ordre précis (tourniquet - round-robin) en utilisant le principe du «passage du bâton».

Le programme 36 montre comment utiliser un sémaphore pour implanter une barrière.

Programme 32 : Exemple de programme sans sémaphore JR

```
import edu.ucdavis.jr.JR;
public class sem1
{
    private static int valeur = 0;

    static process Calculator((int id = 0; id < 50; id++))
    {
        for(int i = 0; i < 5; i++)
        {  valeur = valeur + 2; }
    }
    public static void main(String[] args)
    {
        try
        {
            JR.registerQuiescenceAction(fin);
        }
        catch (edu.ucdavis.jr.QuiescenceRegistrationException e)
        {
             e.printStackTrace();
        }
    }
    public static op void fin()
    {   System.out.println(valeur); }
}
```

Programme 33 : Exemple de création d'un tableau de sémaphore

```
import edu.ucdavis.jr.JR;

cap void () t[] = new cap void() [5];
for (int i = 0; i < 5 ; i++)
{
    t[i] = new sem; // sémaphores initialisés à 0 par défaut
}
...
P(t[1]);
..
V(t[1]);
```

Programme 34 : Exemple de création d'un tableau de sémaphore

```
import edu.ucdavis.jr.JR;

cap void () t[] = new cap void() [5];
cap void () mutex[] = new cap void() [5];
int [] tinit = {2,3,4,5,6}

for (int i = 0; i < 5 ; i++)
{
    t[i] = new sem(tinit[i]);
    mutex[i] = new sem(1);
}
```

Programme 35 : Exemple d'utilisation d'un tableau de sémaphore

```
import edu.ucdavis.jr.JR;

public class Tourniquet
{
    private static final int N = 20;   // nombre de processus
    private static final int C = 4; // nombre de tours
    private static cap void () mutex[] = new cap void()[N];

    static // initialisation du vecteur de sémaphore
    {
        mutex[0] = new sem(1);
        for (int i = 1; i < N ; i++)
        mutex[i] = new sem;  // initialisation à 0
    }
    private static process p((int i = 0 ; i < N; i++))
    {
        for (int tour = 1; tour <+ C; tour ++)
        {
            // section non critique
            // ...
            // Section critique
            P(mutex[i]);      // Entrée en SC
            //.....
            System.out.println("Fil no. " + i);
            V(mutex[(i+1)%N]); // On quitte la section critique
            // Section non critique
            // ....
        }
    }
    public static void main(String [] args){}
}
```

Programme 36 : Implantation une barrière

```
import edu.ucdavis.jr.JR;
public class Barriere
{
   private static final int N = 20;   // nombre de processus
   private static sem done = 0;
   private static cap void () continuer[] = new cap void()[N];

   static // initialisation du vecteur de sémaphore
   {
      for (int i = 0; i < N ; i++)
          continuer[i] = new sem;  // initialisation à 0
   }
   private static process travailleur((int i = 0 ; i < N; i++))
   {
      while (true)
      {
        // Tâche pour l'itération avant la barrière
        // ...
        // Barrière
        V(done);       // indiquer qu'on a fini une itération
        P(continuer[i]); // attendre que tout le monde ait fini
        // Prochaine étape du calcul
      }
   }
   private static process coordonnateur
   {
     int cpt = 0;
     while (true)
     {
        for (int w = 0; w < N ; w++) { P(done); }
        System.out.println("tout le monde est là.. ");
        for (int w = 0; w < N ; w++) { V(continuer[w]); }
        System.out.println("..............tout le monde repart.. ");
     }
   }
   public static void main(String [] args){}
}
```

8 Moniteur

JR ne supporte pas directement les moniteurs. Toutefois il fournit un pré-processeur qui transforme un fichier qui contient le code du moniteur et du code JR en un programme contenant seulement du code JR. Le pré-processeur s'appelle «`m2jr`» et vient avec la distribution de JR. Le fichier contenant le code du moniteur doit posséder l'extension «`.m`». La commande `m2jr` traduit donc un fichier avec une extension «`.m`» en plusieurs fichiers contenant du code JR (un pour le moniteur et un pour les variables de type condition).

Tous les énoncés ou mots clés utilisés pour définir un moniteur commencent par «_». Les mots clés à utiliser dans un moniteur sont :

- «`_monitor`» : pour déclarer un moniteur ;
- «`_proc`» : pour ajouter une méthode dans le moniteur qui sera exécutée en exclusion mutuelle ;
- «`_return`» : pour retourner une valeur à partir du moniteur ;
- «`_condvar`» : pour déclarer une variable de type condition ;
- «`_var`» : pour déclarer une variable globale ;
- «`_wait/_signal`» : pour manipuler les variables de type condition ;
- «`_signal_all`» : pour réveiller tous les processus en attente (fonctionne seulement avec le mode Signal & Continue) ;
- «`_empty`» : pour vérifier s'il y a un processus en attente sur la variable condition ;
- «`_wait(condvar, int priority)`» : pour faire attendre un processus avec une certaine priorité. La file d'attente sur la variable condition devient une file avec priorité. Vous ne pouvez utiliser le `_wait` normal et le `_wait` avec priorité sur la même variable condition ;
- «`_minrank`» : pour retourner la priorité minimale sur la variable condition.

De plus, les moniteurs de JR fournissent différents types de signaux. On peut choisir le type de signal désiré par une option de la commande `m2jr`. Les options sont :

- Signal & Continue (-sc) : avec cette option, l'émetteur du signal continue son exécution et le processus signalé continuera éventuellement son exécution dans le futur. Il devra compétitionner avec tous les processus pour l'exclusion mutuelle. C'est la version «`signal/notify`» vue dans le cours.
- Signal & Wait (-sw) : avec cette option, l'émetteur est bloqué et le processus signalé poursuit immédiatement son exécution. L'exclusion mutuelle est transférée au processus signalé. C'est la version «`signal/wait`» vue dans le cours.

— Signal & Urgent Wait (-su) : cette option est identique à SW mais l'émetteur a la garantie qu'il s'exécutera immédiatement après le processus signalé.

— Signal & Exit (-sx) : avec cette option, l'émetteur est sorti du moniteur et le processus signalé peut s'exécuter immédiatement.

Par défaut, c'est l'option «`-sc`» qui est utilisée.

Le programme 37 donne une exemple d'utilisation des moniteurs.

Programme 37 : Exemple de moniteur en JR

```
 _monitor MoniteurTest
{
    _condvar c1;
    _condvar c2;

    _var int var1;
    _var int var2;

    _proc void test()
    {
      _wait(c1);
      //Faire le traitement prevu
      _signal(c2);
    }
    _proc int testB()
    {
        _return 1;
    }
}
```

Pour compiler un programme contenant des moniteurs, il faut faire :

— `m2jr MonitorTest.m`

— `jr Moniteurtest ...`

Pour compiler avec le signal de type «`-sw`», il faut faire : «`m2jr -sw MonitorTest.m`».

La commande «`m2jr MonitorTest.m`» va créer deux fichiers : «MonitorTest.jr» et «c_m_condvar.jr».

L'option «`-sc`» crée un problème de vol de signal. Par exemple, dans le programme 38, qui gère un tampon contenant n éléments, on ajoute une boucle autour du «`wait`» pour éviter ce problème.

S'il n'y avait pas de boucle, la situation suivante pourrait se produire :

1. Le processus 1 essaie de consommer mais le tampon est vide. Il attend sur «`pasVide`.
2. Le processus 2 dépose un élément, émet un signal pour réveiller le processus 1 et conserve l'exclusion mutuelle.
3. Le processus 3 obtient l'exclusion mutuelle avant le processus 1 et consomme le nouvel élément.

Programme 38 : Exemple du producteur/consommateur

```
_monitor TamponFini
{
   private static final int N = 5; //Taille du tampon
   _var String[] tampon = new String[N];
   _var int debut;
   _var int fin;
   _var int compteur;
   _condvar pasPlein;
   _condvar pasVide;
   _proc void deposer(String donnee)
   {
      while(compteur == N)  _wait(pasPlein);
      tampon[debut] = donnee;
      fin = (fin + 1) % N;
      compteur++;
      _signal(pasVide);
   }
   _proc String retirer()
   {
      while(compteur == 0)  _wait(pasVide);
      String resultat = tampon[debut];
      debut = (debut + 1) % N;
      compteur--;
      _signal(pasPlein);
      _return resultat;
   }
}
```

4. Le processus 1 qui a été réveillé par le signal, tente de consommer l'élément. S'il y a une boucle, il se bloque de nouveau. Sinon il y a un problème.

Si on utilise l'option «`sw`», la solution au problème du tampon ressemble au programme 39.

Dans les deux cas, l'utilisation du tampon se fait comme illustré au programme 40. La sortie produite par le programme est illustrée à la figure 5.

9 Interactions inter-processus

JR offre de multiples techniques pour faire communiquer deux processus. Il est donc possible avec JR de communiquer avec du passage de messages asynchrone, des rendez-vous et des appels de procédures éloignés.

9.1 Passage de messages asynchrone

En Java, la principale façon de passer des messages est d'utiliser les «*Socket*» ou «RMI». En JR, le concept de passage de messages est plus intégré. On utilise des opérations comme des files de messages et les commandes `send` et `receive` pour manipuler les messages. La commande `send` est asynchrone (non-bloquante) et la commande `receive` est synchrone (bloquante).

Programme 39 : Exemple du producteur/consommateur

```
_monitor TamponFini
{
    private static final int N = 5; //Taille du tampon
    _var String[] tampon = new String[N];
    _var int debut;
    _var int fin;
    _var int compteur;
    _condvar notFull;
    _condvar pasVide;
    _proc void deposer(String data)
    {
        if compteur == N)  _wait(pasPlein);
        tampon[fin] = data;
        fin = (fin + 1) % N;
        compteur++;
        _signal(pasVide);
    }
    _proc String retirer()
    {
        if (compteur == 0)  _wait(pasVide);
        String result = tampon[debut];
        debut = (debut + 1) % N;
        compteur--;
        _signal(pasPlein);
        _return result;
    }
}
```

Programme 40 : Exemple du producteur/consommateur

```
public class ProdCons
{
   private static final int N = 12; //Nombre de producteurs et de
    consommateurs
   private static TamponFini monTampon = new TamponFini("Exemple de tampon
    ave Moniteur"); //Le moniteur
   public static void main(String... args){}

   private static process Producteur((int i = 0; i &lt; N; i++))
   {
      monTampon.deposer("Producteur" + i);
   }
   private static process Consommateur((int i = 0; i &lt; N; i++))
   {
      System.out.println("Consommateur " + i + " : " + monTampon.retirer())
   ;
   }
}
```

```
Consommateur10 : Producteur0
Consommateur0 : Producteur1
Consommateur1 : Producteur2
Consommateur2 : Producteur4
Consommateur3 : Producteur5
Consommateur4 : Producteur3
Consommateur8 : Producteur7
Consommateur11 : Producteur10
Consommateur6 : Producteur8
Consommateur5 : Producteur6
Consommateur7 : Producteur11
Consommateur9 : Producteur9
```

FIGURE 5 – Sortie du producteur/consommateur

Comme la communication est asynchrone, les opérations impliquées ne doivent pas être implantées (aucune méthode) et ne doivent pas avoir de valeurs de retour. La portée des opérations est la même que celle des méthodes. Ainsi si une opération est `static` dans un classe `A`, un énoncé `send` sur cette opération peut être servie par tous les processus qui peuvent avoir accès à cette opération (dans la portée).

Une opération de ce type implante en fait un canal de communication. Le programme 41 montre comment déclarer une tel canal. Le programme 42 montre comment envoyer ou recevoir un message sur ce canal en utilisant les mots clés `send` et `receive`.

Programme 41 : Syntaxe pour la déclaration d'un canal ce communication

```
private static op void channel(int);
```

Programme 42 : Syntaxe de JR pour le passage de message

```
private static op void channel(int);
...
send channel(12);
...
int x;
receive channel(x);
```

Le programme 43 utilise la communication par messages asynchrone.

Programme 43 : Exemple de passage de messages en JR

```
import edu.ucdavis.jr.JR;
public class msg1
{
   private static op void channel(int);
   private static process pcs1
   {
     for(int i=0; i<10; i++)
     {
        send channel(i);
        System.out.println("Envoi...");
     }
   }
   private static process pcs2
   {
     int x;
     for(int i=0; i<10; i++)
     {
        // JR.nap(10);
        receive channel(x);
        System.out.println(x);
     }
   }
   public static void main(String... args){}
}
```

Les canaux de communication peuvent évidemment avoir plus d'un paramètre. Le programme 44 montre une communication sur un canal avec plus d'un paramètre.

Programme 44 : Exemple de passage de messages en JR

```
import edu.ucdavis.jr.JR;

public class msg3
{
    private static op void request(int x, int y);
    private static op void response(int somme, int difference);
    public static void main(String... args){}

    private static process Client
    {
        send request(33, 22);
        int somme;
        int diff;
        receive response(somme, diff);
        System.out.printf("Somme =  %d, Différence = %d\n", somme, diff);
    }
    private static process Server
    {
        int x;
        int y;
        receive request(x, y);
        send response(x + y, x - y);
    }
}
```

Les messages sont reçus dans l'ordre d'envoi. Il est toutefois possible qu'un message envoyé par le processus 1 (M1) avant celui envoyé par le processus 2 (M2) soit reçu après (M1 est reçu après M2). Cela peut se produire quand les processus s'exécutent sur des ordinateurs différents.

Le programme 45 montre une implantation du problème des producteurs/consommateurs qui utilise les messages.

Programme 45 : Exemple de producteur/consommateur avec message asynchrones

```
public class ProducteurConsommateur
{
    private static final int N = 12; //Nbr de producteurs/consommateurs
    private static op void canal(String); // le canal de communiaction
    public static void main(String... args){}

    private static process Producteur((int i = 0; i < N; i++))
    {
        send canal("Producteur" + i);
    }

    private static process Consommateur((int i = 0; i < N; i++))
    {
        String data;
        receive canal(donnee);
        System.out.println("Consommateur" + i + " : " + donnee);
    }
}
```

9.1.1 Messages et capacités

On peut aussi envoyer et recevoir des messages en utilisant les capacités. Grâce aux capacités, il devient très simple de partager des canaux de communication.

Le prgramme 46 donne un premier exemple d'utilisation des capacités pour la communication par messages.

Programme 46 : Exemple communication avec une capacité

```
public class msg4
{
    private static op void f(double);
    private static op void g(double);

    public static void main(String... args)
    {
       cap void (double) c1, c2;
       // associer les capacités selon le paramètre
       if (args.length > 0) { c1 = f; c2 = g;}
       else                 { c1 = g; c2 = f;}
       // Appeler les fonctions pointées par c1 et c2
       send c1(4.350);
       send c2(8.21);
    }
    private static process pcsF
    {
        double d;
        receive f(d);
        System.out.println("Processus F a recu " + d);
    }
    private static process pcsG
    {
        double d;
        receive g(d);
        System.out.println("Processus G a recu " + d);
    }
}
```

Les capacités peuvent permettent de passer outre la portée d'une opération. Le programme 47 montre comment communiquer avec un canal local grâce à une capacité.

Comme on peut envoyer des capacités dans des messages, ils peuvent faciliter l'implantation d'une architecture client/serveur. Imaginons que nous ayons N clients qui veulent utiliser une ressource partagée par tous. Pour gérer l'accès à la ressource, l'utilisation d'un serveur est requise. Le programme 48 donne un premier exemple d'implantation du problème des clients/serveur.

Le problème avec ce premier exemple est qu'un serveur ne peut distinguer ses clients. S'il envoie un message à `ressource`, n'importe quel client peut recevoir ce message. Nous devons établir un canal unique entre un client et le serveur. Pour cela on peut utiliser un tableau de canaux. Le programme 49 illustre comment utiliser un tableau de canaux. Dans cet exemple, le client passe son identité avec la requête. Le serveur peut donc répondre directement au bon client.

Programme 47 : Exemple de communication avec une capacité locale

```
public class msg5
{
    private static op void obtenirCapacité(cap void (double));
    public static void main(String... args){    }

    private static process pcsF
    {
        op void f (double);
        send obtenirCapacité(f);
        double d;
        receive f(d);
        System.out.println("Processus F a recu " + d);
    }
    private static process pcsG
    {
        cap void (double) y;
        receive obtenirCapacité(y);
        send y(999.999);
    }
}
```

Programme 48 : Exemple de client/serveur avec communication asynchrone

```
// client
op void demande();
op void resource(Resource);
op void liberation(Resource);
----------------------------------------
// reservation de la ressource
send demande();
receive resource(resource);

//utilisation de la ressources

send liberation(resource); // libération de la ressource
----------------------------------------
// Serveur
while (True)
{
  receive demande();
  send resource(resource);
  receive liberation(resource)
}
```

Programme 49 : Client/serveur avec un tableau de canaux

```
public class clientServeur1
{
    private static final int N = 20;   // nombre de clients
    private static op void demande(int, char);
    private static cap void (double) res[] = new cap void(double)[N];
    static // initialisation du tableau
    {
        for (int i = 0; i < N ; i++)
        res[i] = new op void (double);
    }
    private static process client((int i = 0 ; i < N; i++))
    {
        // ...
        send demande(i, 't');
        // travail en attendant la réponse
        double d;
        receive res[i](d);
        System.out.println("Client " + i + " a recu " + d);
        // ....
    }
    private static process serveur
    {
        int id; char donnee; double reponse;
        while (true)
        {
            receive demande(id, donnee);
            System.out.println("Serveur recoit demande de " + id);
            // Faire une certaine traitement
            reponse = 210;
            send res[id](reponse);
        }
    }
    public static void main(String [] args){}
}
```

Un problème avec cette dernière approche est que le nombre de clients doit être fixe. Pour régler ce problème chaque client peut créer un canal localement et passer une capacité vers le canal dans le message destiné au serveur. Le programme 50 montre comment utiliser les capacités pour implanter les clients et le serveur.

Programme 50 : Client/serveur avec des capacités

```
public class clientServeur2
{
    private static final int N = 20;   // nombre de clients
    private static op void demande(int, cap void (double), char);
    private static process client((int i = 0 ; i < N; i++))
    {
        op void resultat(double);
        // ...
        send demande(i, resultat, 't');
        // travail en attendant la réponse
        double d;
        receive resultat(d);
        System.out.println("Client " + i + " a recu " + d);
        // ....
    }
    private static process serveur
    {
        int id; char donnee; double reponse;
        cap void (double) canalRetour;
        while (true)
        {
            receive demande(id, canalRetour, donnee);
            System.out.println("Serveur recoit demande de " + id);
            // Faire une certaine traitement
            reponse = 210;
            send canalRetour(reponse);
        }
    }
    public static void main(String [] args){}
}
```

Finalement, le problème avec cette version est que si le serveur attend pour une requête, il ne peut attendre pour le message de libération de la ressource et vice-versa. Une solution possible est d'avoir plusieurs processus dans le serveur. Cependant, on doit synchroniser les processus. Une autre solution consiste à créer une opération qui donne des informations sur le type de la demande. Le programme 51 illustre comment implanter ce concept. Ce code fonctionne avec un nombre de processus quelconque et peut être adapté à un environnement réseau avec quelques modifications mineures.

9.2 Rendez-vous

Comme la communication asynchrone, les rendez-vous sont implantés avec les opérations qui servent de file de messages. Pour établir un rendez-vous, on utilise la commande `send` et la commande `inni` (plutôt que `receive`.) Le programme 52 montre comment utiliser l'énoncé `inni` .

Programme 51 : Client/serveur avec communication asynchrone

```
enum Type {REQUEST, RELEASE};
// on suppose l'existence d'un type Ressource
//-----------------------------------------------
// Client
op void demande(Type, cap void (Ressource), Ressource);
cap void (Ressource) canal = new op void (Ressource);

send request(Type.REQUEST, canal, null);
Resource ressource;
receive canal(ressource);
send request(Type.RELEASE, noop, ressource);
//-----------------------------------------------
// Serveur :
cap void (Ressource) client;
Type type;
Ressource ressource;
Queue<Ressource> ressources = new LinkedQueue<Ressource>();
//Remplissage de la file de ressources
Queue<cap void (int)> clients = new LinkedQueue<cap void int>();

while(true)
{
    receive demande(type, client, resource);
    if(type == Type.REQUEST)
    {
        if (resources.isEmpty())  clients.add(client);
        else                      send client(resource.pop());
    }
    else
    {
        if (clients.isEmpty()) resources.put(resource);
        else                   send clients.pop()(resource);
    }
}
```

Programme 52 : Syntaxe pour les rendez-vous

```
int x;
int y;
send commande(2, 3);
receive(x, y);
--------------------------------
inni op_commande_1
{
   // code pour la commande 1
}
[] op_commande_2
{
    //code pour la commande 2
}
```

Une «op_commande_i» est une opération sur laquelle le processus se bloque en attente d'une commande. Elle est de la forme :

```
type_retour op_exp(args) st expr_sync by expr_planif
```

où :

— type_retour : indique le type de la valeur de retour de l'opération ;

— op_exp : donne le nom de l'opération ou de la capacité ;

— args : sont les paramètres de l'opération ;

— `st expr_sync` : ajoute une condition à l'opération (expression de synchronisation). La condition indique quel est le message qui peut être reçu.

— `by expr_planif` : indique l'ordre dans lequel les messages seront servis (expression de planification). Cela doit être un nombre. Le message avec la plus basse valeur selon l'expression de planification sera servie en premier.

S'il n'y a aucune expression de synchronisation ni de planification, les appels (`send`) seront servis dans l'ordre d'arrivée. S'il y a une expression de synchronisation, le premier appel à être servi sera le plus ancien selon l'expression. S'il y a une expression de planification le premier appel à être servi sera celui qui a la valeur minimale selon l'expression. S'il n'y a aucun message qui satisfait toutes les conditions, la réception du message est reportée jusqu'à l'arrivée d'un message satisfaisant les conditions.

On peut ajouter un `else` à l'énoncé de réception. La partie `else` sera exécutée s'il n'y a aucun message qui satisfait les conditions. Ainsi un énoncé de réception avec un `else` ne sera jamais retardé. Le format d'un énoncé `inni` avec un `else` est en montré dans le programme 53.

Programme 53 : Syntaxe de l'énonce «inni» avec un else

```
inni op_command
{
    // enonces
}
...
[] else
{
    // enonces du else
}
```

Les programmes 54 et 55 implante un serveur qui retourne la somme de deux nombres après la réception d'un message. Le premier utilise la communication par messages asynchrones et le second le concepts de rendez-vous.

Le programme 56 donne un autre exemple de serveur utilisant le concept de rendez-vous. Le programme 57 implante un serveur gérant plusieurs ressources et utilise les expressions

Programme 54 : Serveur utilisant la communication asynchrone

```
public class Calculatrice
{
    private static op void demande(int x, int y);
    private static op void reponse(int addition, int sub);

    public static void main(String... args){}

    private static process Client
    {
        send demande(33, 22);
        int somme;
        int difference;
        receive reponse(somme, difference);
        System.out.printf("Somme %d Difference %d", somme, difference);
    }

    private static process Serveur
    {
        int x; int y;
        receive demande(x, y);
        send reponse(x + y, x - y);
    }
}
```

Programme 55 : Serveur utilisant le concept de rendez-vous

```
public class Calculatrice
{
    private static op int calculer(int x, int y);

    public static void main(String... args){}

    private static process Client
    {
         System.out.printf("Somme %d", calculer(33, 22));
    }
    private static process Server
    {
        inni int compute(int x, int y)
        {
            return x + y;
         }
    }
}
```

de synchronisation. Finalement, le programme 58 implante un serveur qui gère un tampon contenant N éléments.

Programme 56 : Exemple de rendez-vous

```
public class rendezVous2
{
    private static op void f(int);
    private static op double g(double);

    private static process pcs1
    {
        int y = 6;
        // ...
        call f(y);
        // ....
    }
    private static process pcs2
    {
        double w;
        // ...
        w =  g(3.14);
        System.out.println("Pcs2 a recu " + w);
        // ....
    }
    private static process serveur
    {
        int z=0;
        for (int i = 0; i < 2 ; i++)
        {
            inni
               void f(int x) { z += x; }
            []
               double g(double u) { return u * u - z; }
        }
    }
    public static void main(String [] args){}
}
```

Programme 57 : Exemple de rendez-vous avec expression de synchronisation

```
int nb_disponible = M;
while (true)
{
    inni
       void demande() st nb_disponible > 0
       { nb_disponible--;}
    []
       void release()
       { nb_disponible ++; }
}
```

Si une opération retourne une seule valeur, il n'est pas nécessaire d'utiliser l'énoncé `call`. Ce dernier est implicite. Si l'opération ne retourne aucune valeur, vous pouvez utiliser l'énoncé `call` (ce n'est pas nécessaire toutefois). La syntaxe est de l'énoncé est :

```
call op_command(args);.
```

Programme 58 : Exemple de rendez-vous avec expression de synchronisation

```
public class rendezVous3
{
   private static op void deposer(int);
   private static op int retirer();
   private static final int N = 10;

   private static process serveur
   {
      int [] tampon = new int [10];
      int compte = 0, debut = 0, fin = 0;
      while (true)
      {
         inni
            void deposer(int element) st compte < N
            {
              tampon[fin] = element;
              fin = (fin +1) % N;
              compte ++;
            }
         []
            int retirer() st compte > 0
            {
              int element = tampon[debut];
              debut =(debut+1) % N;
              compte --;
              return element;
            }
      }
   }
   private static process producteur((int i=0; i< 5; i++))
   {
        for(int j=0; j< 5; j++)   call deposer(z);
   }

   private static process concommateur((int i=0; i< 5; i++))
   {
      int element;
      for(int j=0; j< 5; j++)
      {
        element = retirer();
        System.out.println("Comsommateur " + i + " a retire " + element);
      }
   }

   public static void main(String [] args){}
}
```

Le programme 59 montre que le `receive` est seulement une forme abrégée de l'énoncé `inni`.

Programme 59 : «receive» vs «inni»

```
int x;
int y;
receive op_command(x,y)
---------------------------------
inni void op_command(int a, int b)
{
  x = a;
  y = b;
}
```

Le programme 60 montre que l'énoncé `inni` peut aussi être utilisé pour servir un groupe d'opérations dans une vecteur.

Programme 60 : Exemple de programme JR

```
cap void (int) operations = new cap void (int)[12];
//Fill the array
inni ((int i = 0; i <=  12; i++)) operations[i](int x)
{
    // code...
}
```

En plus de l'énoncé `return`, on peut utiliser deux autres énoncée dans un `inni` :

— `reply` : retourne une valeur à l'appelant mais ne termine pas l'énoncé `inni`. De cette façon vous pouvez toujours faire des opérations dans l'énoncé `inni` mais vous ne pouvez plus retourner de valeur.

— `forward` : délègue la réponse à une autre opération. L'autre opération retournera une valeur à l'appelant et l'énoncé `inni` peut continuer son exécution mais ne peut plus retourner de valeur.

Le programme 61 présente une nouvelle version d'un serveur pour l'allocation des ressources.

Il est aussi possible d'utiliser le `send` au lieu d'un appel car le serveur peut aussi bien répondre à un qu'à l'autre. Toutefois il est impossible de recevoir une valeur de retour avec l'utilisation d'un `send`.

10 Machines virtuelles

Une machine virtuelle JR est une machine virtuelle Java additionnée d'une couche pour le langage JR. Tous les exemples réalisés jusqu'à maintenant fonctionne sur une seule machine

Programme 61 : Exemple de serveur avec rendez-vous

```
import java.util.*;
public class rendezVous1
{
   private static final int N = 25; //Nbr de clients
   private static final int I = 25; //Nbr d'itérations

   public static void main(String... args){}
   private static op Resource request();
   private static op void release(Resource);

   private static process Client((int i = 0; i <= N; i++))
   {
      for(int a = 0; a <= I; a++)
      {
         Resource resource = request();
         System.out.printf("C %d avec R %d \n", i, resource.getId());
         call release(resource);
      }
   }

   private static process Server
   {
      Queue<Resource> resources = new LinkedList<Resource>();

      for(int i = 1; i <=5; i++)
         resources.add(new Resource(i));

      while(true)
      {
         inni
           Resource request() st !resources.isEmpty()
           {
               return resources.poll();
           }
         []
          void release(Resource resource)
          {
              resources.add(resource);
          }
      }
   }
   private static final class Resource
   {
      private final int id;
      private Resource(int id)
      {
          this.id = id;
      }
      private int getId()
      {
          return id;
      }
  }
}
```

virtuelle. Nous allons voir comment déclarer et utiliser plusieurs machines virtuelles en JR. Ces machines virtuelles peuvent aussi s'exécuter physiquement sur différents ordinateurs.

Une fois que l'on a créé plusieurs machines virtuelles, on peut spécifier sur quel ordinateur un objet sera créé avec une variante de l'opérateur «`new`. Par la suite, tout le développement est transparent. Par exemple, une opération «`send`» destinée à un processus sur une autre machine virtuelle se fera de la même façon qu'un «`send`» sur la même machine virtuelle à quelques détails près.

Un élément important est que chaque machine virtuelle contient une partie statique. Chaque partie statique est locale à la machine virtuelle. Il faut être très prudent.

Le programme 62 donne la syntaxe pour la création d'une machine virtuelle. Il créé une machine virtuelle, nommée `vm1`, sur le même ordinateur que l'exécution courante du code. Le programme 63 montre comment créer une machine virtuelle sur un autre ordinateur. Pour créer une machine virtuelle distante sur un ordinateur M1, il faut pouvoir s'y connecter avec ssh (ou autre) sans mot de passe. JR ne fournit aucun moyen pour détruire une machine virtuelle. Comme tous les objets, la machine virtuelle est collecté par le ramasse-miette quand elle n'est plus utilisée (plus référencée ou lorsqu'elle est inoccupée i.e. tous les processus sont terminés ou bloqués).

Programme 62 : Syntaxe pour la création de machine virtuelle

```
vm vm1 = new vm();
```

Programme 63 : Exemple de création de machine virtuelle

```
vm vm2 = new wm() on "192.168.1.200"; //IP Adresse of the machine
vm vm3 = new wm() on "pc12"; //Name of the machine
vm vm4 = new vm() on vm3; //The machine where vm3 is located
```

Pour créer un objet sur une machine virtuelle, il faut utiliser le mot clé `remote`. Le programme 64 illustre l'utilisation de cet énoncé. Par défaut, l'objet est créé sur la machine virtuelle courante. Le programme 65 montre comment le créer sur une autre machine virtuelle. le programme 66 montre qu'il est possible de combiner la création d'un objet éloigné et de la machine virtuelle sur laquelle il s'exécutera. Il est important de noter que la classe de tout object éloigné doit être déclaré publique.

Programme 64 : Syntaxe pour la création de machine virtuelle éloignée

```
remote Person person = new remote Person();
```

Lorsque l'on manipule un objet éloigné, deux nouveaux champs prédéfinis sont accessibles :

— this.remote : retourne la référence éloigné de l'objet courant.

Programme 65 : Syntaxe pour la création de machine virtuelle

```
remote Person person = new remote Person() on vm1;
```

Programme 66 : Syntaxe pour la création de machine virtuelle

```
remote Person person = new remote Person() on new vm() on "localhost";
```

— vm.thisvm : retourne la machine virtuelle sur laquelle l'objet courant a été créé.

Lorsque l'on travaille sur une seule machine virtuelle, les paramètres sont toujours passés par valeur. Lorsque l'on travaille avec des références éloignés sur plusieurs machines virtuelles, une copie de l'objet passé en paramètre est créé et passé à la destination (JR utilise RMI et la sérialisation pour y parvenir)

Il y a d'autres aspects intéressants à savoir lorsque l'on travaille avec plusieurs machines virtuelles. En premier, `System.out` et `System.in` sont hérités de la machine virtuelle initiale. Les impressions se font donc toujours sur la machine initiale mais l'ordre d'impression ne sera pas toujours le même (non-déterministe). De plus, la première machine virtuelle s'exécute dans le «répertoire courant» mais les autres débutent leur exécution dans le répertoire de base de l'utilisateur (Home).

Le programme 67 donne un exemple d'utilisation des machines virtuelles. On reprend le problème du producteur/consommateur en utilisant la communication par message asynchrones. Dans cet exemple on crée une producteur et un consommateur sur deux machines virtuelles distinctes. Tous les consommateurs sont créés sur la même machine virtuelle et tous les producteurs sur une autre. Les deux machines virtuelles sont créées sur le même ordinateur. Il serait simple toutefois de les créer sur des ordinateurs distincts.

11 Installation de JR

11.1 Installion de JR sur Windows

11.1.1 Préalable

Pour installer JR, il faut tout d'abord avoir ou installer Java.

11.1.2 Installation

1. Télécharger JR
   (`http://web.cs.ucdavis.edu/~olsson/research/jr/#JR_implementation`)
2. Décompresser l'archive à l'endroit où vous désirez l'installer
   (supposons `C:\Program Files\JR\`)

Programme 67 : Exemple de producteur/consommateur

```
class ProducteurConsommateur {
    private static final int N = 12; //Nb producteurs/consommateurs
    private static op void deposit(String); //canal de communication

    public static void main(String... args)
    {
        vm vmConsommateur = new vm();
        vm vmProducteur = new vm();

        for(int i = 0; i &lt; N; i++)
        {
            new remote Consommateur() on vmConsommateur;
            new remote Producteur() on vmProducteur;
        }
    }
    public static class Consommateur
    {
        private static process Consommateur
        {
            String data;
            receive deposit(data);
            System.out.println("Consommateur " + data);
        }
    }
    public static class Producteur
    {
        private static process Producteur
        {
            send deposit("Producteur");
        }
    }
}
```

3. Il faut configurer des variables d'environnement et des constantes :
   - JR_HOME=C :\Program Files\JR
   - PATH=%PATH% ;%JR_HOME%\bin\
   - CLASSPATH=. ;%JR_HOME%\classes\jrt.jar ;%JR_HOME%\classes\jrx.jar
   - JRSH=cmd
   - JRSHC=/C.

11.1.3 Vérification de l'installation

Pour tester l'installation, il faut :

1. Ouvrir une console (Start menu -> Execute -> cmd)
2. Aller dans le dossier de JR (`cd %JR_HOME%`)
3. Lancer le test : cd vsuite ; ../jrv/jrv quick

Cela exécutera plusieurs programmes en plus d'afficher des informations sur le système. Il devrait afficher des résultats similaires au suivant :

```
C:\Program Files\JR\vsuite>..\jrv\jrv quick
Starting JRV
JR_HOME= C:\Program Files\JR\
JRC=     perl "C:\Program Files\JR\/bin/jrc"
JRRUN=   perl "C:\Program Files\JR\/bin/jrrun"
JAVAC=   "C:\Program Files\Java\jdk1.6.0_17/bin\javac.EXE"
JAVA=    "C:\Program Files\Java\jdk1.6.0_17/bin\java.EXE"
ccr2jr=  perl "C:\Program Files\JR\/bin/ccr2jr"
csp2jr=  perl "C:\Program Files\JR\/bin/csp2jr"
m2jr=    perl "C:\Program Files\JR\/bin/m2jr"
WHICH=   perl "C:\Program Files\JR\/bin/which.pl"
CMP=     perl "C:\Program Files\JR\/bin/cmp.pl"
GREP=    perl "C:\Program Files\JR\/bin/grep.pl"
SORT=    perl "C:\Program Files\JR\/bin/sort.pl"
TAIL=    perl "C:\Program Files\JR\/bin/tail.pl"
jr compiler version "2.00602 (Mon Jun 1 10:59:20 PDT 2009)"
jr rts      version "2.00602 (Mon Jun 1 10:59:25 PDT 2009)"
HOST= DESKTOP-PC
Start Directory= C:\Program Files\JR\/vsuite
JR.JRT = C:\Program Files\JR\/classes/jrt.jar
-rw-rw-rw-   1 0        0          2090324 Jun  1  2009 C:\Program Files\JR\/classes/jrt.jar
JR.JRX = C:\Program Files\JR\/classes/jrx.jar
-rw-rw-rw-   1 0        0           227198 Jun  1  2009 C:\Program Files\JR\/classes/jrx.jar
Operating System= Windows_NT
original CLASSPATH= .;C:\Program Files\JR\classes\jrt.jar;C:\Program Files\JR\classes\jrx.jar
jrv sets CLASSPATH= .;C:\Program Files\JR\classes/jrt.jar;C:\Program Files\JR\classes/jrx.jar
DATE= Thu Jan 14 19:01:35 2010
quick/baby:
quick/fact_2:
quick/misc_invocation_count_st_by_0:
DATE= Thu Jan 14 19:01:45 2010
Elapsed time (hh:mm:ss)= 00:00:10
```

Pour démarrer plus de tests, il faut faire la commande : ../jrv/jrv

Cette commande exécutera plusieurs tests qui prendront beaucoup de temps et nécessiteront RSH dans certains cas.

11.2 Installation de JR sur Linux (Ubuntu)

1. Téléchargement de JR : tar file (jr.tar), gzipped tar file (jr.tar.gz), or zipped file (jr.zip).
2. Copier le fichier dans le répertoire d'installation (ex. : /usr/local/src).
3. Décompresser et désarchiver le fichier. Cela va placer tous les fichiers dans le répertoire "jr".

   Le répertoire (ex. : /usr/local/src/jr) servira à initialiser la variable d'environnement «JR_HOME».

   Vous devez utiliser la commande «sudo» pour avoir le droit d'installer.
4. JR utilise java et perl...

   (a) Il faut installer Java 1.8 (openJDK) : «sudo apt-get install openjdk-8-jdk».

   Si vous avez besoin d'un Java plus récent, vous devrez installer les deux en parallèles et ajuster les chemins de recherche de JR.

   (b) Il faut installer Perl

   Il faut s'assurer qu'il est accessible (doivent être dans le search path, ie PATH).

   L'implantation courante de JR assume que perl est dans /usr/bin, ce qui est correct pour Linux (Ubuntu).

   Si vous utiliser un autre système, il faut changer certains fichiers dans «$JR_HOME/bin» et «$JR_HOME/jrv». Pour ce faire,exécuter la commande suivante dans «$JR_HOME» : `tweak/tweakall` «*pathname_de_perl*» ou «`tweak/tweakall 'which perl'`.
5. Initialiser les variables d'environnement

   (a) JR utilise par défaut csh. Pour changer, par exemple pour utiliser bash, cela il faut faire :

   ```
   export JRSH=/bin/bash
   ```

   (b) Pour trouver les commandes reliées à JR, il faut configurer les variables suivantes :

   ```
   JR_HOME=/usr/local/src/jr; export JR_HOME
   PATH=$PATH:$JR_HOME/bin; export PATH
   export JRRSH=/usr/bin/ssh
   export CLASSPATH=.:$JR_HOME/classes/jrt.jar:$JR_HOME/classes/jrx.jar
   export DISPLAY=:0.0
   ```