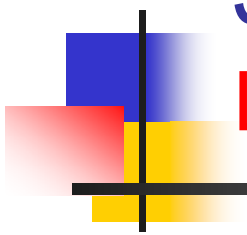
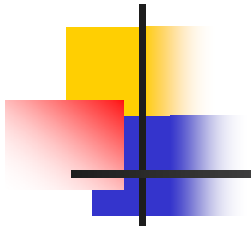


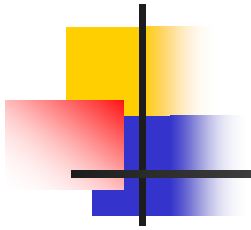
Java
License Profesionnelle CISI 2009-2010

Cours 12 : les threads



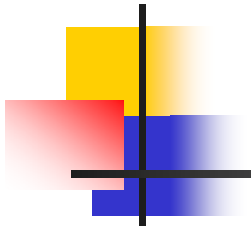
Introduction

- Qu'est ce qu'un Thread ?
  - Actuellement, toutes les machines qu'elles soient mono ou multiprocesseurs permettent d'exécuter plus ou moins simultanément plusieurs programmes
    - On parle encore de tâches ou de processus
  - Sur les machines monoprocess, la simultanéité est une illusion
    - Un seul programme utilise les ressources de la machine mais l'environnement "passe la main" d'un programme à l'autre pour gérer les attentes ou la simultanéité
  - Java
    - présente l'originalité d'appliquer cette possibilité de multiprogrammation au sein d'un même programme dont on dit qu'il est formé de plusieurs threads indépendants
    - Ces threads peuvent facilement communiquer entre eux et partager des données



Introduction

- Exemple introductif
  - Voici un programme qui va lancer 3 threads simples, chacun d'entre eux se contentant d'afficher un certain nombre de fois un texte donné
    - 10 fois « bonjour » pour le premier
    - 12 fois « bonsoir » pour le deuxième
    - 5 fois un changement de ligne pour le 3ème thread



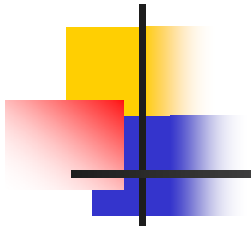
Introduction

- Exemple introductif (suite 1)
  - En Java, un thread est un objet qui dispose d'une méthode qui s'appelle run qui sera exécutée lorsque le thread sera démarré
  - Deux façons de créer cette classe
    - La plus simple consiste à créer une classe dérivée de la classe Thread

```
class Ecrit extends Thread
    {public Ecrit(String texte, int nb)
        {           this.texte=texte;
                    this.nb=nb;
        }
    Public void run ()
        { for(int i=0; i<nb; i++)
                    System.out.print(texte);
        }
```

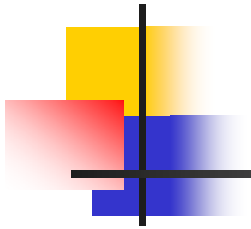
    - La deuxième est d'utiliser l'interface Runnable (voir plus loin)
  - La création des objets threads pourra se faire depuis n'importe quel endroit du programme, par exemple, depuis le main, de cette façon :

```
Ecrit e1 = new Ecrit ("bonjour",10);
Ecrit e1 = new Ecrit ("bonjour",10);
Ecrit e1 = new Ecrit ("bonjour",10);
```



Introduction

- Exemple introductif (suite 2)
  - Ces appels ne font cependant rien d'autre que de créer des objets
  - Le lancement se fait par :
    - e1.start(); //lancement du thread 1
  - Cependant, pour assurer une sorte de simultanéité des traitements, il faut provoquer des interruptions
  - L'interruption se fait par la méthode
    - sleep(t)
      - où t est le temps d'attente en multiseconde
  - Cette démarche laisse ainsi la possibilité à d'autres threads de s'exécuter à leur tour
  - La méthode sleep est susceptible de générer une exception de type InterruptedException dont nous verrons plus tard l'intérêt

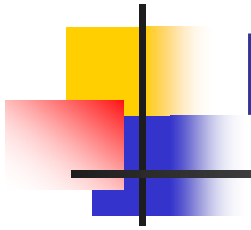


Introduction

- Exemple complet : TstThr1.java

```
public class TstThr1
{ public static void main (String args[])
  { Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
    Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
    Ecrit e3 = new Ecrit ("\n", 5, 15) ;
    e1.start() ;
    e2.start() ;
    e3.start() ;
    Clavier.lireInt();
  }
}
class Ecrit extends Thread
{ public Ecrit (String texte, int nb, long
      attente)
  { this.texte = texte ;
    this.nb = nb ;
    this.attente = attente ;
  }
public void run ()
  {  try
     { for (int i=0 ; i<nb ; i++)
       { System.out.print (texte) ;
         sleep (attente) ;
       }
     }
     catch (InterruptedException e) {}  //
      impose par sleep
  }
  private String texte ;
  private int nb ;
  private long attente ;
}
```

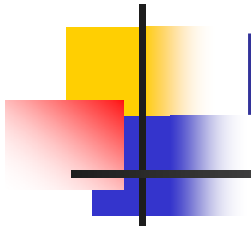
Utilisation de l'interface Runnable



- C'est la deuxième possibilité
  - Pour créer des threads
  - Cette interface contient une méthode run à redéfinir
  - Application à l'exemple précédent
    - Obligé de créer une classe (appelons là encore Ecrit) :

```
class Ecrit implements Runnable
{ public Ecrit (String texte, int nb, long attente)
  { //mêmes instructions que précédemment
  }
   public void run ()
      { //même contenu que précédemment
        //en n'oubliant pas que sleep est statique : l'appel sera :
       //Thread.sleep(t);
    }
}
```

Utilisation de l'interface Runnable



- C'est la deuxième possibilité
  - Nous serons amenés à créer des objets de type Ecrit, par ex :
    - Ecrit e1 = new Ecrit("bonjour", 10,5);
  - Cette fois, ces objets ne sont plus des threads et ne peuvent donc plus être lancés par start
  - On doit d'abord créer des objets de type Thread en utilisant une forme particulière de constructeur recevant en argument un objet implémentant l'interface Runnable, par exemple

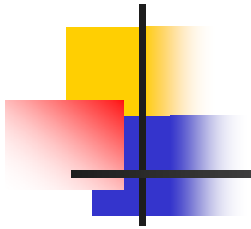
```
Thread t1=new Thread(e1) //crée un objet thread associé à l'objet e1
                         //qui doit implémenter l'interface Runnable
                         //pour disposer d'une méthode run
```

  - On lance ensuite classiquement ce thread par start
  - t1.start;

Utilisation de l'interface Runnable

- Exemple précédent transformé : TstThr3

```
public class TstThr3
{ public static void main (String args[])
  { Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
    Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
    Ecrit e3 = new Ecrit ("\n", 5, 15) ;

    Thread t1 = new Thread (e1) ; t1.start() ;
    Thread t2 = new Thread (e2) ; t2.start() ;
    Thread t3 = new Thread (e3) ; t3.start() ;
  }
}
class Ecrit implements Runnable
{ public Ecrit (String texte, int nb, long
      attente)
  { this.texte = texte ;
    this.nb = nb ;
    this.attente = attente ;
  }
public void run ()
  {  try
    { for (int i=0 ; i<nb ; i++)
      { System.out.print (texte) ;
        Thread.sleep (attente) ;  // attention
     Thread.sleep
      }
    }
    catch (InterruptedException e) {}  //
     impose par sleep
  }
  private String texte ;
  private int nb ;
  private long attente ;
}
```



Interruption d'un thread

- Démarche usuelle
  - Dans les exemples précédents, les threads s'achevaient tout naturellement avec la fin de l'exécution de leur méthode run
  - Dans certains cas, on peut avoir besoin d'interrompre prématurément un thread depuis un autre thread
    - On utilise la méthode interrupt de la classe Thread
  - Par ailleurs, dans un thread, il est possible de connaître l'état d'un thread

Thread 1

```
t.interrupt(); //positionne un
               //indicateur dans t
```

Thread 2 nommé t

```
run
{...
   if(interrupted)
     {...
            return; //fin du thread
     }
}
```

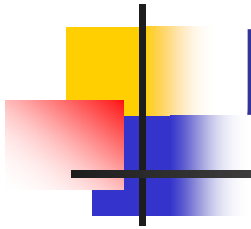
Interruption d'un thread

- Exemple complet : TstInter

```
public class TstInter
{ public static void main (String args[])
  { Ecrit e1 = new Ecrit ("bonjour ", 5) ;
    Ecrit e2 = new Ecrit ("bonsoir ", 10) ;
    Ecrit e3 = new Ecrit ("\n", 35) ;
    e1.start() ;
    e2.start() ;
    e3.start() ;
    Clavier.lireString();
    e1.interrupt() ;     // interruption premier
      thread
    System.out.println ("\n*** Arret premier
      thread ***") ;
    Clavier.lireString();
    e2.interrupt() ;     // interruption deuxième
      thread
    e3.interrupt() ;     // interruption troisième
      thread
    System.out.println ("\n*** Arret deux
      derniers threads ***") ;
  }
}
```

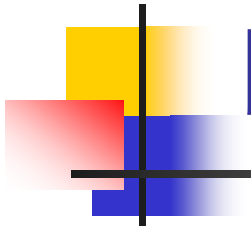
```
class Ecrit extends Thread
{ public Ecrit (String texte, long attente)
  { this.texte = texte ;
    this.attente = attente ;
  }
public void run ()
  { try
    {  while (true)     // boucle infinie
       { if (interrupted()) return ;
         System.out.print (texte) ;
         sleep (attente) ;
       }
    }
    catch (InterruptedException e)
    { return ;   // on peut omettre return ici
    }
  }

  private String texte ;
  private long attente ;
}
```

Threads démons et arrêt brutal



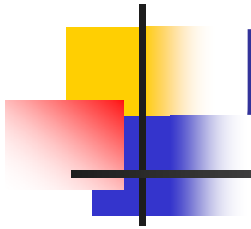
- Principe
  - Un programme est amené à lancer un ou plusieurs threads
  - Jusqu'ici, nous avons considéré qu'un programme se terminait lorsque le dernier thread le constituant était arrêté
  - En réalité il existe 2 catégories de thread
    - Les threads dits utilisateurs
    - Les threads dits démons

Threads démons et arrêt brutal



- Thread démon
  - La particularité d'un thread démon est la suivante
    - Si à un moment donné, les euls threads en cours d'exécution d'un même programme sont des démons, ces derniers sont arrêtés brutalement et le programme se termine
  - Par défaut, un thread est créé dans la catégorie du thread qui l'a créé
    - (utilisateur pour main, donc pour tous les threads, tant qu'on n'a rien demandé d'autre)
  - Pour faire d'un thread un démon
    - on effectue l'appel setDaemon(true) avant d'appeler la méthode start

Threads démons et arrêt brutal

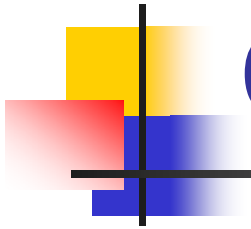


- Adaptation de l'exemple précédent
  - Nous avons fait des trois threads des threads démons
    - Le thread principal (main) s'arrête lorsque l'utilisateur frappe un texte quelconque
    - On constate que les trois threads sont interrompus

Threads démons et arrêt brutal

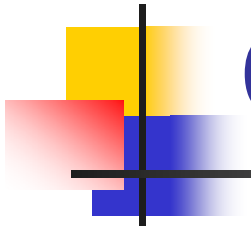
- Adaptation de l'exemple précédent

```
public class Demons
{ public static void main (String args[])
  { Ecrit e1 = new Ecrit ("bonjour ", 5) ;
    Ecrit e2 = new Ecrit ("bonsoir ", 10) ;
    Ecrit e3 = new Ecrit ("\n", 35) ;
    e1.setDaemon (true) ; e1.start() ;
    e2.setDaemon (true) ; e2.start() ;
    e3.setDaemon (true) ; e3.start() ;
    Clavier.lireString() ;  // met fin au main,
      donc ici
                          // aux trois autres threads
      démons
  }
}
class Ecrit extends Thread
{ public Ecrit (String texte, long attente)
  { this.texte = texte ;
    this.attente = attente ;
  }
public void run ()
  { try
    {  while (true)      // boucle infinie
       { if (interrupted()) return ;
         System.out.print (texte) ;
         sleep (attente) ;
       }
    }
    catch (InterruptedException e)
    { return ;   // on peut omettre return ici
    }
  }

  private String texte ;
  private long attente ;
}
```



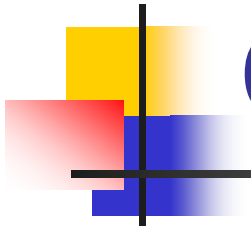
Coordination de Threads

- Principe
  - L'avantage des threads sur les processus, c'est qu'ils appartiennent à un même programme
  - Ils peuvent donc éventuellement partager les mêmes objets
  - Cet avantage s'accompagne parfois de contraintes
  - En effet, dans certains cas, il faut éviter que deux threads puissent accéder en même temps à un objet
    - Ceci sera réglé par les « méthodes synchronisées »
  - Ou encore, un thread doit attendre un autre d'avoir fini un travail sur un objet avant qu'il y accède à son tour
    - Ceci sera réglé par des mécanismes d'attente



Coordination de Threads

- Méthodes synchronisées
  - Rappel : l'environnement peut interrompre un thread (pour donner la main à un autre) à tout moment
  - Exemple de 2 threads répétant indéfiniment les actions suivantes
    1. Incrémentation d'un nombre et calcul de son carré (premier thread)
    2. Affichage du nombre et de son carré (second thread)
  - On voit que si le premier thread se trouve interrompu entre l'incrémentation et le calcul de carré, le second risque d'afficher le nouveau nombre et l'ancien carré
  - Pour pallier cette difficulté
    - Java permet de déclarer des méthodes avec le mot clé synchronized
    - A un instant donné, une seule méthode ainsi déclarée peut être appelée pour un objet donné



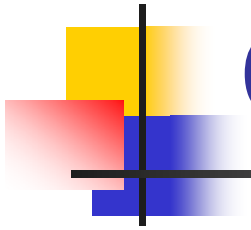
Coordination de Threads

- Exemple :
  - Traitement de l'exemple « nombre et carré »
  - Il s'agit de partager n et son carré entre deux threads
  - Les informations sont regroupées dans un objet nomb de type Nombres
  - Cette classe dispose de deux méthodes synchronisées (mutuellement exclusives)
    - calcul qui incrémente n et calcule la valeur de carre
    - affiche qui affiche les valeurs de n et de carre
  - Nous créons 2 threads de deux classes différentes
    - calc de classe ThrCalc qui appelle, à son rythme (défini par appel de sleep), la méthode calcul de nomb
    - aff de la classe ThrAff qui appelle, à son rythme, la méthode affich de nomb
  - Les deux threads sont lancés par main et interrompus lorsque l'utilisateur le souhaite (en frappant un texte quelconque)

Coordination de Threads

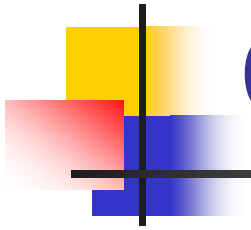
- Exemple : Synchr1

```
public class Synchr1
{ public static void main (String args[])
  { Nombres nomb = new Nombres() ;
    Thread calc = new ThrCalc (nomb) ;
    Thread aff  = new ThrAff (nomb) ;
    System.out.println ("Suite de carres - tapez retour pour
         arreter") ;
    calc.start() ; aff.start() ;
    Clavier.lireString() ;
    calc.interrupt() ; aff.interrupt() ;
  }
}
class Nombres
{ public synchronized void calcul()
  { n++ ;
    carre = n*n ;
  }
  public synchronized void affiche ()
  { System.out.println (n + " a pour carre " + carre) ;
  }
  private int n=0, carre ;
}

class ThrCalc extends Thread
{ public ThrCalc (Nombres nomb)
  { this.nomb = nomb ;
  }
public void run ()
  { try
    { while (!interrupted())
      { nomb.calcul () ;
        sleep (50) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private Nombres nomb ;
}
class ThrAff extends Thread
{ public ThrAff (Nombres nomb)
  { this.nomb = nomb ;
  }
   public void run ()
  { try
    { while (!interrupted())
      { nomb.affiche() ;
        sleep (75) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private Nombres nomb ;
}
```



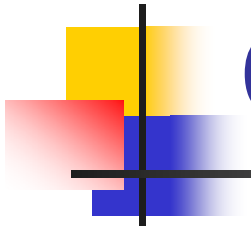
Coordination de Threads

- Exemple : Synchr1
  - 1 a pour carre 1
  - 2 a pour carre 4
  - 4 a pour carre 16
  - 5 a pour carre 25
  - 7 a pour carre 49
  - 8 a pour carre 64
  - 10 a pour carre 100
  - 11 a pour carre 121
  - 13 a pour carre 169
  - 14 a pour carre 196
  - 16 a pour carre 256
  - 17 a pour carre 289



Coordination de Threads

- Notion de verrou
  - À un instant donné, une seule méthode synchronisée peut donc accéder à un objet donné
  - Pour mettre en place une telle contrainte, on peut considérer que, pour chaque objet doté d'au moins une méthode synchronisée, l'environnement gère un « verrou » (ou une clé) unique permettant l'accès à l'objet
    - Le verrou est attribué à la méthode synchronisée appelée pour l'objet et il est restitué à la sortie
    - Tant que le verrou n'est pas restitué, aucune autre méthode synchronisée ne peut le recevoir

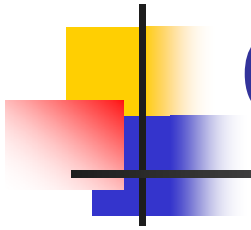


Coordination de Threads

- Notion de verrou
  - Ce mécanisme d'exclusion mutuelle est basé sur l'objet lui-même et non sur le thread
  - Cette distinction peut s'avérer importante dans une situation comme celle-ci

```
void synchronized f(...) //on suppose f appelée sur un objet o
      {... //partie I
                  g() //appel de g sur le même objet o
      ... //partie II
      }
void g(...)
{...
}
```

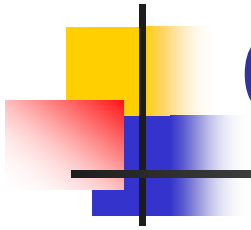
  - La méthode f (synchronisée), appelée sur un objet o, appelle g (non synchronisée) sur le même objet
  - Le verrou de o, attribué au début à o, est rendu lors de l'appel de g
  - g peut alors se trouver interrompue par un thread qui peut modifier o, ainsi rien ne garantit que o ne sera pas modifié entre l'exécution de partie I et celle de partie II. En revanche, cette garantie existerait si g était elle aussi synchronisée



Coordination de Threads

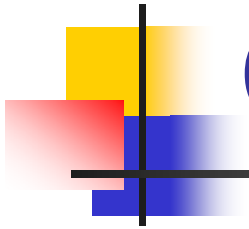
- L'instruction synchronized
  - Une méthode synchronisée acquiert donc le verrou sur l'objet qui l'a appelée pour toute la durée de son exécution
  - L'utilisation d'une méthode synchronisée comporte deux contraintes
    - L'objet concerné (celui sur lequel elle acquiert le verrou) est nécessairement celui qui l'a appelée
    - L'objet est verrouillé pour toute la durée de l'exécution de la méthode
  - L'instruction synchronized permet d'acquérir un verrou sur un objet quelconque pour une durée limitée à l'exécution d'un simple bloc

```
Synchronized (objet)
{ //instructions
}
```



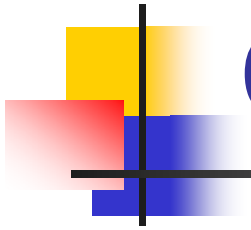
Coordination de Threads

- Interblocage
  - L'utilisation de verrous sur des objets peut conduire à une situation de blocage, connue sous le nom d' "étreinte mortelle" qui peut se définir ainsi
    - Le thread t1 possède le verrou de l'objet o1 et il attend le verrou de l'objet o2
    - Le thread t2 possède le verrou de l'objet o2 et il attend le verrou de l'objet o1
  - Cette situation est à éviter car Java n'est pas en mesure de la traiter



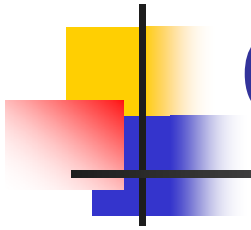
Coordination de Threads

- Attente et notification
  - Il arrive des situations où l'on doit coordonner l'exécution des threads : attente d'un thread...
  - Java offre un mécanisme basé sur l'objet et sur les méthodes synchronisées que nous venons d'étudier :
    - Une méthode synchronisée peut appeler la méthode *wait* de l'objet dont elle possède le verrou, ce qui a pour effet :
      - de rendre le verrou à l'environnement qui pourra, l'attribuer à une autre méthode synchronisée
      - de mettre en attente le thread correspondant; plusieurs threads peuvent être en attente sur un objet
    - Une méthode synchronisée peut appeler la méthode *notifyAll* d'un objet pour prévenir tous les threads en attente sur cet objet et leur donner la possibilité de s'exécuter



Coordination de Threads

- Exemple 1 : gestion d'une « réserve »
  - Il comporte
    - Un thread qui ajoute une quantité donnée
    - Deux threads qui puisent chacun une quantité donnée
  - Un thread ne peut puiser dans la réserve que si elle contient une quantité suffisante
  - La réserve est représentée par un objet r, de type Reserve
  - Cette classe dispose de deux méthodes synchronisées *puise* et *ajoute*
    - Lorsque *puise* s'aperçoit que la réserve est insuffisante, elle appelle *wait* pour mettre le thread correspondant en attente
    - Parallèlement, *ajoute* appelle *notifyAll* après chaque ajout
  - Les 3 threads sont lancés par main et interrompus lorsque l'utilisateur le souhaite (en frappant un texte)



Coordination de Threads

Exemple 1 : Synchro3

```
public class Synchro3
{ public static void main (String args[])
  { Reserve r = new Reserve () ;
    ThrAjout ta1 = new ThrAjout (r, 100, 15) ;
    ThrAjout ta2 = new ThrAjout (r, 50,  20) ;
    ThrPuise tp =  new ThrPuise  (r, 300, 10) ;
    System.out.println ("Suivi de stock --- faire entree pour
        arreter ") ;
    ta1.start () ; ta2.start () ; tp.start () ;
    Clavier.lireString() ;
    ta1.interrupt () ; ta2.interrupt () ; tp.interrupt () ;
  }
}
class Reserve extends Thread
{ public synchronized void puise (int v) throws
        InterruptedException
  { if (v <= stock) { System.out.print ("-- on puise " + v) ;
            stock -= v ;
            System.out.println (" et il reste " + stock ) ;
  }
   else  { System.out.println ("** stock de " + stock
             + " insuffisant pour puiser " + v ) ;
              wait() ;
         }
  }
```

```
public synchronized void ajoute (int v)
  { stock += v ;
    System.out.println ("++ on ajoute " + v  + " et il y a
        maintenant " + stock) ;
    notifyAll() ;
  }
  private int stock = 500 ;    // stock initial = 500
}
class ThrAjout extends Thread
{ public ThrAjout (Reserve r, int vol, int delai)
  { this.vol = vol ; this.r = r ; this.delai = delai ;
  }
  public void run ()
  { try
    { while (!interrupted())
      { r.ajoute (vol) ;
        sleep (delai) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private int vol ;
  private Reserve r ;
  private int delai ;
}
```

Coordination de Threads

- Exemple 1 : Synchro3 (suite)

```
class ThrPuise extends Thread
{ public ThrPuise (Reserve r, int vol, int delai)
  { this.vol = vol ; this.r = r ; this.delai = delai ;
  }
  public void run ()
  { try
    { while (!interrupted())
      { r.puise (vol) ;
        sleep (delai) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private int vol ;
  private Reserve r ;
  private int delai ;
}
```

- Résultat

```
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
++ on ajoute 100 et il y a maintenant 150
** stock de 150 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 200
** stock de 200 insuffisant pour puiser 300
++ on ajoute 100 et il y a maintenant 300
-- on puise 300 et il reste 0
** stock de 0 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 50
++ on ajoute 100 et il y a maintenant 150
** stock de 150 insuffisant pour puiser 300
++ on ajoute 100 et il y a maintenant 250
** stock de 250 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 300
++ on ajoute 100 et il y a maintenant 400
-- on puise 300 et il reste 100
** stock de 100 insuffisant pour puiser 300
++ on ajoute 50 et il y a maintenant 150
++ on ajoute 100 et il y a maintenant 250
** stock de 250 insuffisant pour puiser 300
```

Coordination de Threads

- Exemple 2 : retour sur l'exemple avec calc et aff
  - Les deux threads calc et aff n'étaient pas coordonnés
  - On pouvait incrémenter plusieurs fois le nombre avant qu'il n'y ait d'affichage ou encore afficher plusieurs fois les mêmes informations
- Ici
  - On va faire en sorte que malgré leurs rythmes différents, les deux threads soient coordonnés, c.à.d qu'on effectue alternativement une incrémentation et un calcul
  - Pour ce faire, on utilise wait et notifyAllainsi qu'un indicateur booléen prêt permettant aux deux threads de communiquer entre eux

- Synchr4

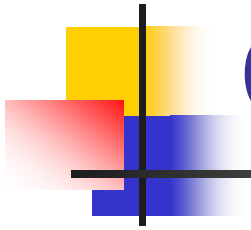
```
public class Synchr4
{ public static void main (String args[])
  { Nombres nomb = new Nombres() ;
    Thread calc = new ThrChange (nomb) ;
    Thread aff  = new ThrAff (nomb) ;
    System.out.println ("Suite de carres - tapez retour
       pour arreter") ;
    calc.start() ; aff.start() ;
    Clavier.lireString() ;
    calc.interrupt() ; aff.interrupt() ;
  }
}
class Nombres
{ public synchronized void calcul() throws
       InterruptedException
  { if (!pret)
    { n++ ;
      carre = n*n ;
      pret = true ;
      notifyAll() ;
    }
    else wait() ;
  }
catch (InterruptedException e) {}
  }
```

Coordination de Threads

- Exemple 2 (suite)

```
public boolean pret ()
  { return pret ;
  }
  private int n=1, carre ;
  private boolean pret = false ;
}
// on aurait pu garder les tests sur pret

class ThrChange extends Thread
{ public ThrChange (Nombres nomb)
  { this.nomb = nomb ;
  }
  public void run ()
  { try
    { while (!interrupted())
      { nomb.calcul() ;
        sleep (5) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private Nombres nomb ;
}
```

```
class ThrAff extends Thread
{ public ThrAff (Nombres nomb)
  { this.nomb = nomb ;
  }
   public void run ()
  { try
    { while (!interrupted())
      { nomb.affiche() ;
        sleep (2) ;
      }
    }
    catch (InterruptedException e) {}
  }
  private Nombres nomb ;
}
```



Coordination de Threads

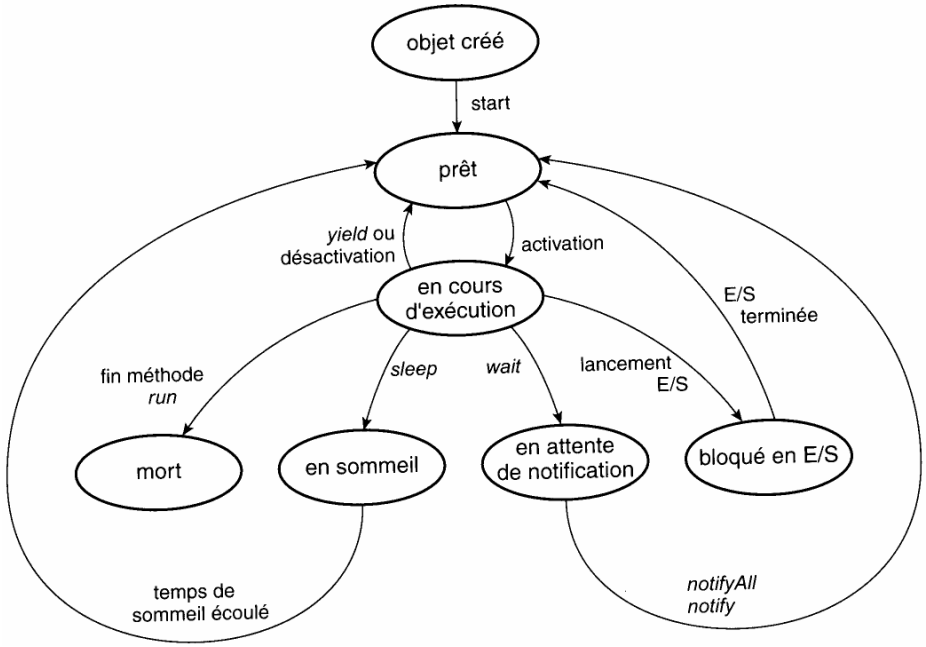
- Exemple 2 (suite 2)

Résultat

- 56 a pour carre 3136
- 57 a pour carre 3249
- 58 a pour carre 3364
- 59 a pour carre 3481
- 60 a pour carre 3600
- 61 a pour carre 3721
- 62 a pour carre 3844
- 63 a pour carre 3969
- 64 a pour carre 4096
- 65 a pour carre 4225
- 66 a pour carre 4356
- 67 a pour carre 4489
- 68 a pour carre 4624
- 69 a pour carre 4761
- 70 a pour carre 4900
- 71 a pour carre 5041
- 72 a pour carre 5184
- 73 a pour carre 5329
- 74 a pour carre 5476
- 75 a pour carre 5625

États d'un thread

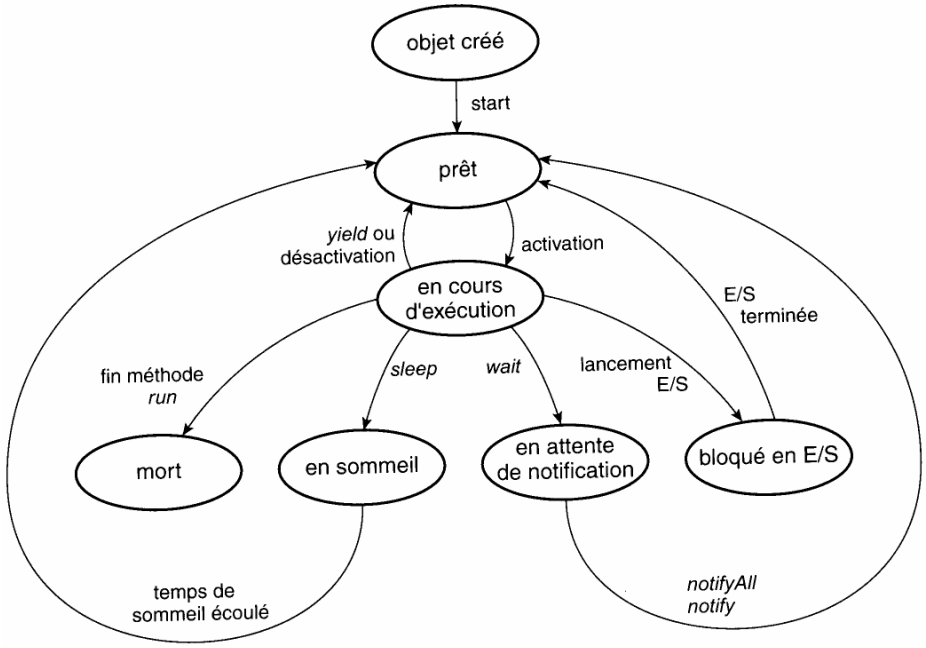
- Principe
  - Nous avons vu comment un thread peut se mettre en « attente » ou en « sommeil »
  - On va faire ici le point sur les différents états dans lesquels peut se trouver un thread et sur les actions qui les font passer d'un état à un autre



Les différents états d'un thread

États d'un thread

- Les états
  - Au départ, on crée un objet thread
  - Tant que l'on ne fait rien, il n'a aucune chance d'être exécuté
  - L'appel de start rend le thread disponible pour l'exécution (il est considéré comme prêt)
  - L'environnement peut faire passer le thread de l'état prêt à l'état en cours d'exécution (c'est le système qui décide)
  - Un thread en cours d'exécution peut subir différentes actions :
    - interrompu et revenir à l'état prêt
    - Mis en sommeil par appel de sleep...



Les différents états d'un thread