Université de Nice - Sophia Antipolis
Faculté des Sciences

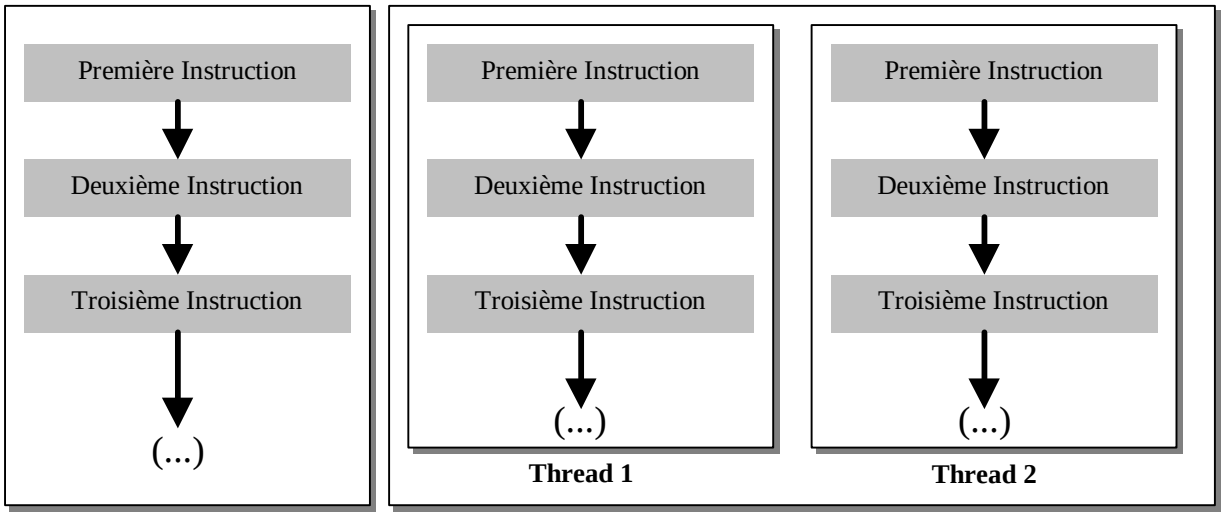
DEUG MIAS MP1

Programmation 2000-01

12. PROGRAMMATION CONCOURANTE

A. Introduction ou comment être au four et au moulin.

Avec l'introduction aux interfaces graphiques et à la librairie AWT, nous avons été amenés à penser en termes de programmation événementielle. En effet au lieu d'envisager notre programme comme une séquence de fonctions et procédures régie par des instructions de contrôle telles que If, Switch, ou For, nous avons du envisager l'ensemble des événements possibles (click de bouton, mouvement dans une barre de défilement, rafraîchissement de l'affichage...) et pour chacun d'eux il nous a fallu programmer une réponse sans savoir s'ils se produiraient et, le cas échéant, quand ils se produiraient. L'interface graphique est aussi un exemple frappant de la nécessité de pouvoir faire plusieurs choses à la fois. Par exemple : ' écouter ' ce qui se passe à la souris ou au clavier tout en continuant d'afficher une animation. Ce besoin d'isoler et d'exécuter en parallèle plusieurs tâches trouve sa réponse avec les ' Thread ' (à prononcer *Srèd* ☺). Le mot ' Thread ' signifie en Anglais ' fil '. Fil de Nylon, fil de la conversation... dans notre cas il représente un fil d'exécution c'est pourquoi il est parfois appelé 'contexte d'exécution' ou encore 'processus léger'. Il permet d'introduire dans un programme des séquences parallèles d'instructions et donc de faire plusieurs choses à la fois. Si cette nouvelle option est très séduisante et effectivement très puissante, elle implique une certaine discipline et hygiène de programmation car on a déjà parfois du mal à savoir d'où viennent nos erreurs dans une séquence unique d'instruction, imaginez maintenant qu'il y en ait plusieurs en même temps... les Thread impliquent de penser en termes de processus concourants, cohabitant et souvent même collaborant.



Le programme principal est lui même un Thread. Les Thread ne créent pas des programmes indépendants, mais s'exécutent en parallèle au sein d'un même programme et bénéficient ainsi d'un environnement commun où ils partagent les ressources (CPU, mémoire...) et en particulier peuvent partager une même variable. Ceci à des avantages, notamment le fait de partager une même variable est un moyen pour deux Thread de s'échanger des données, et des inconvénients, il est en particulier souvent nécessaire de s'assurer que deux Thread ne sont pas en conflits lors d'un accès à une ressource, ce qui demande alors d'être capable de bloquer ou synchroniser les Thread souhaitant l'accès.

B. Anatomie d'un Thread.

Hormis une classe particulière de Thread (les Démons) que nous n'aborderons pas dans ce cours, chaque Thread a un cycle de vie au cours duquel il est tenu de :

- **Ne pas être égoïste** (à moins d'avoir de bonnes raisons) : ne pas monopoliser les ressources et en particulier laisser aux autres Thread l'opportunité de vivre leur vie en ne s'accaparant pas le microprocesseur.

- **Savoir mourir quand l'heure a sonné** : prévoir la condition ou le mécanisme d'arrêt du Thread et ne pas tourner sans raison en rond indéfiniment jusqu'à une mort brutale qui pourrait intervenir à un instant critique de l'exécution du Thread (ex: sauvegarde de données).

Un programme ne s'arrête que lorsque tous ses Thread (qui ne sont pas des démons) se sont arrêtés, lui y compris bien entendu. La classe Thread est la façon la plus simple de créer un Thread :

- On crée un Thread en déclarant une classe qui étend la classe `java.lang.Thread`
- On surcharge la méthode `run()` avec une nouvelle méthode `run()` implantant les instruction que l'on désire exécuter dans le Thread. La méthode `run()` joue un peu le même rôle pour un Thread que la méthode `main(String[])` pour le programme principal.
- On oublie pas de laisser la main aux autres Thread par exemple en se mettant en sommeil régulièrement en utilisant la méthode `sleep(long)`. La méthode `sleep(long)` fait dormir le processus pendant le nombre (entier long) de millisecondes, passé en argument. Cette méthode peut renvoyer des exceptions (objets décrivant une erreur lorsqu'elle survient) appelées `InterruptedException` lorsque le Thread est interrompu dans son sommeil par quelque chose (un autre Thread, le programme principal...). Dans un premier temps nous ne ferons qu'attraper ces exception avec `try{} catch(Exception) {}` et nous les ignorerons.

La classe Talker étend la classe Thread pour être exécutable en parallèle.

Un constructeur avec en paramètre le mot à afficher.

Surcharge de la méthode pour programmer la procédure principale du Thread.

Dormir entre chaque tour pour ne pas être égoïste.

```
class Talker extends Thread
  {
    private String myWord = null;

    public Talker(String p_Word) { myWord=p_Word; }

    public void run()
      {
        for(int l_meter=0;l_meter<10;l_meter++)
          {
            try { sleep(500); } catch (InterruptedException p_Exception) { }
            System.out.println(myWord);
          }
        System.out.println("End of " + myWord);
      }
  }

public class YesNo
  {
    static public void main(String[] p_Arg)
      {
        Talker l_Talker1 = new Talker("Yes");
        Talker l_Talker2 = new Talker("No");
        Talker l_Talker3 = new Talker("May be");

        l_Talker1.start();
        l_Talker2.start();
        l_Talker3.start();
      }
  }
```

Création de 3 Thread

Lancement des 3 Thread. La méthode start() est héritée de la classe Thread

Exercice 5.1 Tapez et étudiez l'exemple donné. Modifiez le code pour passer un paramètre au constructeur du Thread. Ce paramètre sera le nombre de millisecondes à utiliser dans le `sleep(long)`. Dans le programme principale, adaptez la création des Thread pour qu'ils aient comme temps de sommeil 200, 400, 600. Regardez l'influence sur le résultat et faites des essais en changeant ces valeurs pour bien comprendre ce qui se passe. Notez comment le `sleep(long)` influence la répartition du temps d'exécution.

Exercice 5.2 Programmez un petit jeu de vitesse de frappe. Vous utiliserez deux classes étendant la classe Thread :

`class Killer`
Qui attend 4 secondes avant d'afficher "\nToo late ! :(" et de terminer le programme.

`class Typer`
Qui affiche "You have four seconds to type your name...", lit le nom au clavier, affiche "The winner is... *le_nom* ! ;)" et termine le programme

La classe principale `QuickTyper` créera dans sa procédure `main(String[])` un Thread de chaque type et les exécutera.

Java ne permet pas le muti-héritage: une classe ne peut étendre qu'une seulle autre classe. Cependant on peut souhaiter par exemple qu'un objet soit à la fois une fenêtre et un Thread, pour que la fenêtre ait sa propre vie. On utilise alors une autre façon d'écrire des Threads qui est l'implantation de l'interface Runnable dans une classe:

```
public class AClassOfThread implements Runnable
  {

    public void run() {  ...   }

    public void start()
      {
        Thread l_NewThread = new Thread(this);
        l_NewThread.start();
      }
  }
```

C. Tous les Thread ne naissent pas égaux en droit.

La plupart des ordinateurs ont seulement une unité centrale de traitement (un microprocesseur ou CPU), ainsi les Thread doivent partager l'unité centrale de traitement avec les autres Thread. Nous avons vu que la méthode `sleep()` permet d'endormir un Thread au profit d'un autre. Il existe un autre moyen de gérer le passage du contrôle d'un Thread à un autre: à chaque Thread de Java est associée une priorité c'est à dire une valeur numérique qui permet de comparer l'importance de deux Thread et de donner une préférence d'exécution à l'un ou à l'autre. Lorsqu'un Thread est créé, il hérite de la priorité du Thread qui le crée. Cette priorité peut être modifiée en utilisant la méthode `setPriority(int)` en donnant une valeur entre MIN_PRIORITY et MAX_PRIORITY (constantes définies dans la classe Thread). Pour des Thread candidats à l'exécution ayant la même priorité, JAVA fait tourner le droit d'exécution d'un Thread à l'autre dès que ce droit est libéré et à condition qu'il n'y ait pas un Thread de priorité supérieure. La priorité est principalement utilisée pour gérer l'efficacité d'une application.

Un Thread donné peut, à tout moment, renoncer à son droit d'exécution en appelant la méthode `yield()`. Contrairement à `sleep(long)`, ce renoncement n'est pas une mise en sommeil pour une durée fixée au profit de tous les Threads, mais seulement un passage temporaire du contrôle aux autres Thread de même priorité. Notons que `yield()` n'émet pas d'exceptions, nous n'avons donc pas besoin de l'entourer d'une close `try & catch`.

Exercice 5.3 Reprenez le programme utilisé dans l'exercice 5.1 tel qu'il était donné en exemple avant d'être modifié. Modifiez le code du Thread pour qu'il ne s'endorme qu'une milliseconde. Dans le programme principal associez à chacun des Thread trois priorités différentes (MIN_PRIORITY, MIN_PRIORITY+2, MIN_PRIORITY+4) avant de les lancer. Etudiez le résultat sachant que chaque Thread est programmé pour dormir le même temps.

Exercice 5.4 Modifiez l'exemple obtenu précédemment pour que les processus ne s'endorment pas mais passent la main aux processus de même priorité. Regarder le résultat obtenu, essayez avec deux des Thread ayant la même priorité, puis avec tous les Thread à priorité égale.

D. Touche pas à mon Thread !

Le deuxième aspect de la d'une programmation non égoïste des Thread est de gérer la synchronisation des accès à une même ressource. Pour illustrer ce problème regardez le code donné en exemple ci dessous.

```
class Meter
  {
    private long myRacerOne = 0;
    private long myRacerTwo = 0;

    public long getRacerOne() { return(myRacerOne); }
    public long getRacerTwo() { return(myRacerTwo); }
    public void set(long p_RacerOne, long p_RacerTwo)
      {
        myRacerOne = p_RacerOne;
        myRacerTwo = p_RacerTwo;
      }
}
```

Classe qui gère un double compteur.

```
class Incrementer extends Thread
  {
    private Meter myMeter = null;

    public Incrementer(Meter p_Meter) { myMeter=p_Meter; }

    public void run()
      {
        while(true){myMeter.set(myMeter.getRacerOne()+1,myMeter.getRacerTwo()+1);}
      }
  }
```

Ce Thread incrémente un double compteur. L'incrémentation étant identique on peut intuitivement penser que les deux valeurs seront toujours les mêmes

```
class Checker extends Thread
  {
    private Meter myMeter = null;

    public Checker(Meter p_Meter) { myMeter=p_Meter; }

    public void run()
      {
        while(true)
          {
            try { sleep(10); } catch (InterruptedException p_Exception) { }
            if (myMeter.getRacerOne()!=myMeter.getRacerTwo())
              {
                System.out.println("Different");
              }
          }
      }
  }
```

Ce Thread vérifie que les deux compteurs sont à la même valeur. Si l'incrémentation est faite par le Thread précédent, on peut penser que ce Thread n'affichera jamais la chaîne "Different"

```
public class RunningMeter
  {
    static public void main(String[] p_Arg)
      {
        Meter l_Meter = new Meter();
        Incrementer l_Incrementer = new Incrementer(l_Meter);
        Checker l_Checker = new Checker(l_Meter);
        l_Incrementer.start();
        l_Checker.start();
      }
  }
```

Classe principale crée deux Thread de chaque type, on s'attend donc à avoir un programme qui tourne en boucle infinie sans jamais rien afficher

Exercice 5.5 Programmez, compilez et lancez cet exemple. Que constatez-vous ?
Utilisez **Crtl-C** pour arrêter le programme (oui je sais, c'est pas propre).
Maintenant la question est : comment expliquez vous le résultat obtenu ?

En fait dans cet exemple, il arrive que le Thread `Checker` se réveille et fasse son test exactement entre les deux instructions `myRacerOne = p_RacerOne;` et `myRacerTwo = p_RacerTwo;`. A cet instant précis, les deux valeurs sont différentes. On voit ici poindre un des aspects de conception les plus compliqué de la programmation concourante: les conflits d'utilisation d'une ressource (mémoire, périphérique...)

Pour palier à ce problème, le langage JAVA propose la directive de synchronisation `synchronized` permettant de créer des blocs d'instructions ne pouvant être interrompus. Deux types de synchronisation existent:

1, La synchronisation des méthodes d'un objet:

```
public synchronized void set(long p_RacerOne, long p_RacerTwo)
      {
        myRacerOne = p_RacerOne;
        myRacerTwo = p_RacerTwo;
      }
```

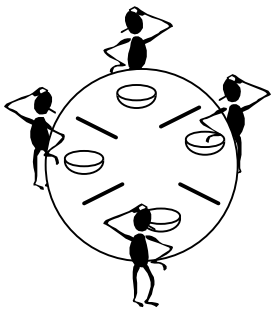
Qui permet de bloquer l'accès à cet objet par toute méthode synchronisée. Donc aucun autre thread ne pourra appeler une méthode synchronisée d'un objet Meter si un autre Thread est entrain d'utiliser une méthode synchronisée de ce même objet. Un Thread essayant d'accéder à l'objet serait bloqué jusqu'à ce que l'autre Thread libère l'objet.

2, La synchronisation d'un bloc d'instructions conditionnellement à la disponibilité d'un objet:

```
public void run()
  {
    while(true)
      {
        try { sleep(10); } catch (InterruptedException p_Exception) { }
        synchronized (myMeter)
          {
            if (myMeter.getRacerOne()!=myMeter.getRacerTwo())
              {
                System.out.println("Different");
              }
          }
      }
  }
```

Ici le bloc d'instructions n'est exécuté que si l'objet myMeter n'est pas bloqué par un autre processus. Lorsque l'accès est autorisé, l'objet sera bloqué le temps d'exécuter de bloc d'instructions.

Exercice 5.6 Modifiez l'exemple précédent avec les indications ci-dessus et vérifiez le résultat.



Evidement toute solution amène son lot de nouveaux problèmes, et la synchronisation introduit le problème de boucles sans fin dues à des objets qui attendent chacun que l'autre libère un objet pour finir leur traitement. Une façon très répandue d'illustrer ce problème est de parler du dîner des philosophes: Des philosophes se rencontrent autour d'un dîner chinois. Sur la table il y a un bol de riz devant chaque philosophe et une baguette pour manger entre chaque philosophe. Si les philosophes pensent de manière identique (même algorithme dans la méthode `run()`), chacun va par exemple essayer de se saisir de la baguette à sa droite puis de celle à gauche, manger du riz et les reposer. Si la scène se passe en parallèle, alors il y a de fortes chances pour que chacun se saisisse d'une baguette (celle à sa droite) et attende ad vitam eternam l'autre baguette puisque l'on voit que c'est un cercle vicieux. Afin d'empêcher se genre de situations dans les Thread on aura à cœur d'anticiper les boucles et de prévoir des algorithme `run()` qui évitent ce type de situation. On peut par exemple ici, numéroter les baguettes et faire en sorte qu'un philosophe prenne toujours en premier la baguette de plus petit nombre puis l'autre. Ainsi le philosophe entre la baguette 1 et 2 prend la baguette 1, le philosophe entre la baguette 4 et 1 ne peut rien prendre car la baguette 1 est prise et par conséquent le philosophe entre la baguette 3 et 4 va pouvoir prendre deux baguette, manger, et reposer les deux baguettes faisant ainsi sortir le système d'une boucle sans fin.

Exercice d'approfondissement 5.7 En approfondissement lorsque vous aurez fini ce TP, vous pouvez programmer le dîner des philosophes et vérifier le problème et le résoudre.

E. Tea time: du producteur au consommateur

Synchroniser n'est pas suffisant, il faut aussi pouvoir avertir les autres Thread qu'une ressource n'est plus disponible ou qu'elle est à nouveau disponible: il est inutile de faire tourner des Thread qui attendent après une ressource tant qu'elle n'est pas disponible. Tout objet dispose en particulier de deux méthodes `wait()` et `notifyAll()` permettant de suspendre (`wait`) l'exécution d'un Thread jusqu'à la notification (`notify`) d'un changement.

Voyons ce mécanisme sur un exemple: Imaginez un Tea Time Londonien, où les invités piochent régulièrement dans une boîte de After Eight alors que parallèlement l'hôte remplit la boite à intervalles réguliers. Un objet représentant la boite de chocolats peut se programmer de la façon suivante:

```
class AfterEightBox
  {
    private int myNumberOfAfterEight=0;

    public AfterEightBox(int p_NumberOfAfterEight)
      {
        myNumberOfAfterEight = p_NumberOfAfterEight;
      }

    public synchronized void get()
      {
        if (myNumberOfAfterEight>0) myNumberOfAfterEight--;
        else try
              {
                wait();
              } catch (InterruptedException p_Exception) { }
      }

    public synchronized void put(int p_Number)
      {
        myNumberOfAfterEight+=p_Number;
        notifyAll();
      }
  }
```

Constructeur initialisant le nombre de chocolats

Procédure pour prendre un chocolat

S'il n'y en a plus, le Thread appelant doit attendre

Procédure pour ajouter des chocolats

On prévient tous les Thread en attente que la situation a changé

Les méthodes wait et notify existent avec des variantes (temps maximal d'attente, nombre de thread prévenus...)

Exercice 5.8 En utilisant l'objet écrit ci dessus, réalisez un programme avec trois classes supplémentaires:

```
class Guest extends Thread
 {  ...
    public Guest(String p_Name, AfterEightBox p_Box, int p_Delay) {     ... }
    public void run() {...}
  }
```

La classe des invités, ayant un nom et piochant dans la boite de chocolats à intervalles réguliers en affichant un petit message avec leur nom pour indiquer qu'ils viennent de se servir.

```
class Host extends Thread  {   ...  }
```

La classe de l'hôte qui ne diffère de la classe des invités que par sa méthode `run()` : l'hôte attend régulièrement avant d'ajouter 10 chocolats dans la boîte.

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

La classe principale, qui crée une boite de 10 chocolats, trois invités de noms différents piochant dans la boite respectivement toutes les 700, 1200 et 1500 milisecondes et un hôte qui rajoute des chocolats toutes les 7000 milisecondes. Compilez et exécutez le programme, vous devriez observer des mangeurs plus ou mois rapides qui prennent les dix chocolats et attentent que l'hôte en remette.

F. Le Thread : programmation d'une mort annoncée.

Comme nous l'avons dit au début, un Thread doit savoir mourir proprement et au bon moment. L'un des modèles de programmation de Thread proposé par les concepteur de JAVA et qui s'utilise beaucoup est le suivant :

```
class Person extends Thread
 {
    private boolean myIsAlive = true;
    private String myName = null;

    public Person(String p_Name)
      {
        myName = p_Name;
      }

    public void die()
      {
        myIsAlive=false;
        interrupt();
        System.out.println(myName + " is dead ! " );
      }

    public void run()
      {
        while(myIsAlive)
          {
            System.out.println(myName + " is alive ! " );
            try { sleep(500);} catch (InterruptedException p_Exception) { }
          }
      }
  }
```

Variable indiquant si le Thread est en vie

Méthode permettant de tuer le Thread

On change la valeur de la variable

On interrompt le Thread s'il dormait

Exercice 5.9 En utilisant la classe donnée ci-dessus et la classe killer :

```
class Killer extends Thread
 {
    private int myDelay = 0;
    private Person myTarget = null;

    public Killer(int p_Delay, Person p_Target)
      {
        myDelay = p_Delay;
        myTarget = p_Target;
      }

    public void run()
      {
        try { sleep(myDelay);} catch (InterruptedException p_Exception) { }
        System.out.println("Bhoo ! ");
        myTarget.die();
      }
  }
```

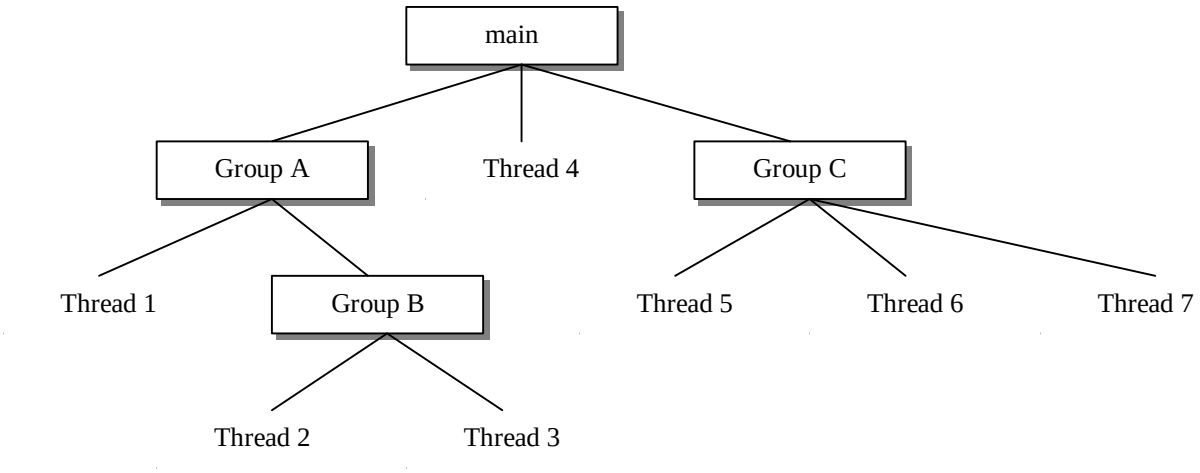
Réalisez un programme créant deux personnes et un tueur par personne ayant respectivement un délai de 3 secondes et 5 secondes. Lancez le programme et suivez les chroniques d'une mort annoncée.

La technique de la variable peut être adaptée en fonction des besoins. Ici par exemple, modifiez la variable et la fonction pour qu'elles soient statiques et que vous puissiez tuer tous les Thread d'une classe d'un seul coup avec un seul tueur. N'oubliez pas de modifier la classe killer en conséquence.

Vous noterez que l'on ne peut plus utiliser la procédure `interrupt()` car il faudrait pouvoir l'appeler sur tous les Thread d'un seul coup. Par conséquent si un Thread est endormi, il faudra attendre qu'il se réveille pour le tuer, et cela peut durer longtemps. Java offre donc une possibilité de grouper les Thread et d'appliquer des méthodes au groupe entier... c'est le sujet de notre prochain chapitre.

G. Une tribu de Thread : un attribut de Thread

La classe ThreadGroup de JAVA permet de contrôler des groupes de Thread dans une même application. Un groupe peut contenir un nombre arbitraire de Thread que l'on a regrouper, par exemple, parce qu'ils font la même chose ou qu'ils doivent être démarrés ou arrêtés en même temps. Un groupe peut non seulement contenir des Threads, mais aussi d'autres groupes, le résultat est un arbre de groupes, le premier groupe (la racine) étant appelé main.



Les Thread comme les groupes de Thread ont des noms qui permettent de les repérer. On peut ainsi créer un nombre indéterminé de Thread et de groupes et les manipuler en parcourant les groupes et l'arbre.

Avec ces nouvelles fonctionnalités, nous pouvons réécrire l'exemple précèdent.

La classe personne utilise maintenant la gestion des noms de Thread héritée de la classe Thread :

```
class Person extends Thread
 {
    private boolean myIsAlive = true;

    public Person(ThreadGroup p_Group, String p_Name)
      {
        super(p_Group,p_Name);
      }

    public void die()
      {
        myIsAlive=false;
        interrupt();
        System.out.println(getName() + " is dead ! " );
      }

    public void run()
      {
        while(myIsAlive)
          {
            System.out.println(getName() + " is alive ! " );
            try { sleep(1000);}
            catch (InterruptedException p_Exception)
              {
                System.out.println(getName() + "  is interrupted" );
              }
          }
      }
  }
```

Constructeur nécessitant un groupe et un nom

On appelle le constructeur de la classe dont on hérite (la classe Thread) en lui passant les paramètres pour créer un Thread nommé et appartenant à un groupe

On affiche un message si un Thread est interrompu pendant son sommeil.

La classe killer est adaptée pour travailler sur un groupe de Thread:

```
class Killer extends Thread
 {
    private int myDelay = 0;
    private ThreadGroup myTargetGroup = null;

    public Killer(int p_Delay, ThreadGroup p_TargetGroup)
      {
        myDelay = p_Delay;
        myTargetGroup = p_TargetGroup;
      }

    public void run()
      {
        try { sleep(myDelay);} catch (InterruptedException p_Exception) { }
        System.out.println("Bhoo ! ");

        int l_NbThreads = myTargetGroup.activeCount();

        Thread[] l_ListOfThreads = new Thread[l_NbThreads];
        myTargetGroup.enumerate(l_ListOfThreads);

        for (int l_Meter = 0; l_Meter < l_NbThreads; l_Meter++)
          ((Person)l_ListOfThreads[l_Meter]).die();

        myTargetGroup.interrupt();
      }
  }
```

Constructeur d'un tueur de groupe

Compter le nombre de Thread dans le groupe

Récupérer les références des Thread dans un tableau de références

Parcourir le tableau de Thread

Pour chaque référence, rappeler au système que c'est une instance de la classe Person *pour pouvoir appeler la méthode* die()

Interrompre les Thread qui dorment

```
public class GroupKiller
 {
    public static void main(String[] args)
      {
        ThreadGroup l_TheGroupOfPerson = new ThreadGroup("Persons from MP1");

        Killer l_Killer = new Killer(5000,l_TheGroupOfPerson);

        (new Person(l_TheGroupOfPerson,"Fabien")).start();
        (new Person(l_TheGroupOfPerson,"Adeline")).start();
        (new Person(l_TheGroupOfPerson,"Thomas")).start();

        l_Killer.start();
      }
}
```

Créer un groupe avec un nom parlant...

Créer un tueur de groupe

Créer des personnes, les ajouter au groupe et les lancer

Lancer le tueur

Exercice 5.10 Tapez l'exemple ci-dessus, compilez et regardez le résultat. Modifiez la classe Person pour pouvoir paramètrer le temps d'assoupissement des Thread et choisissez comme temps dans le programme principal 800, 1000 et 1200. Vous remarquerez que seulement deux Thread sont dérangés dans leur sommeil. Comprenez-vous pourquoi ?

Créez un sous-groupe en introduisant les lignes suivantes dans le programme principal:

```
ThreadGroup l_TheGroupOfPerson2 = new ThreadGroup(l_TheGroupOfPerson,"Lecturers");
(new Person(l_TheGroupOfPerson,"Pr Brunstein",300)).start();
(new Person(l_TheGroupOfPerson,"Pr Klein",1250)).start();
```

Voyez que le tueur les trouvent tous car le nouveau groupe est inclus dans le groupe du tueur.

H. J'ai des fourmis dans le processeur

Les Threads permettent donc de créer et de manipuler des objets évoluants en parallèle (enfin presque, étant donné que sur une machine n'ayant qu'un microprocesseur, il est évident que le parallélisme est simulé et qu'en faite on passe la main à chaque Thread de façon séquentielle). L'une des applications les plus intéressantes est de modéliser des objets du monde réel par des objets informatiques en essayant de programmer le comportement de chacun de façon à pouvoir simuler informatiquement les évolutions du système réel modélisé. En tant que physiciens vous pourriez par exemple imaginer de simuler des particules et leur mouvement Brownien. Dans la même veine, en intelligence artificielle il existe une branche assez récente (début des années 90) qui s'intéresse aux systèmes multi-agent. Un tel système comporte des agents que l'on qualifient de délibératifs (s'ils ont par exemple, un comportement évolué, de la mémoire, un langage et des protocoles de communications, des capacités de planification, des émotions...) ou de réactifs (s'ils sont simples et se contentent de réagir à leur environnement sous forme de réflexes plus ou moins aléatoires). Les exemples les plus parlant d'agents réactifs simulent des comportements de fourmis, d'abeilles, de termites,... pour mettre en évidence que l'interaction d'un nombre élevé de petites unités de programmations élémentaire (par exemple des petits Thread) peut faire émerger des comportement sociaux très complexes et d'une certaine façon intelligents: la recherche de nourriture chez les fourmis tend vers le chemin de longueur minimale malgré les obstacles, les termites construisent des nids d'une architecture très robuste et complexe, les guêpes se répartissent les rôles et se spécialisent dans certaines tâches de façon à se que toute la communauté s'équilibre...

Nous allons ici utiliser tout ce que vous avez appris au cours de cette UV pour afficher graphiquement la simulation du comportement des fourmis fourragères: fourmis en charge de ranger les greniers de la fourmilière en rassemblant les graines en tas.

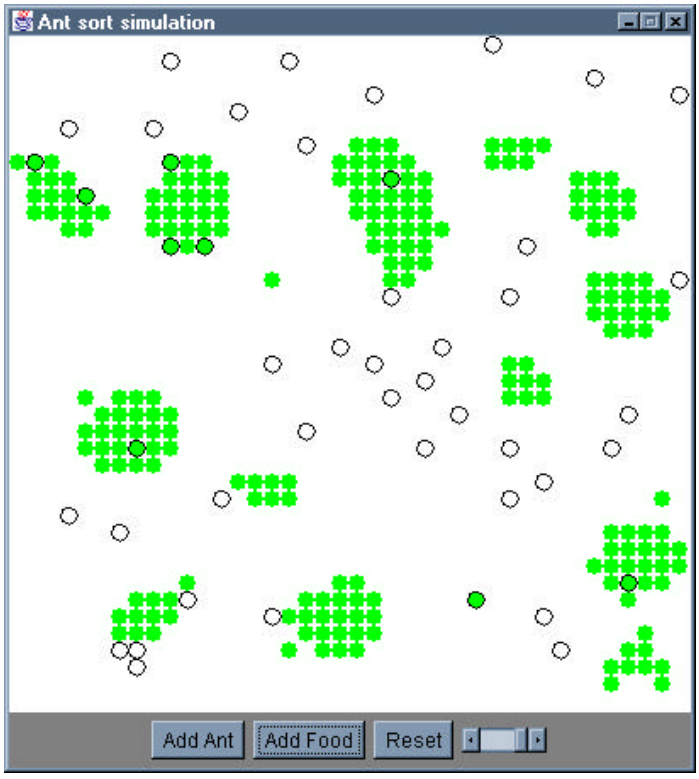
Exercice 5.11 Avant de commencer à s'intéresser à la programmation des fourmis, il nous faut programmer une classe très simple permettant de gérer la position de celles-ci. Préparez la classe suivante:

```
class Position
  {
    int myX=0;
    int myY=0;
    public Position (int p_X, int p_Y) { ...}
    public int getX() { ... }                          } Récupérer les coordonnées
    public int getY() { ... }                          }
    public void setX(int p_X) { ... }                  } Spécifier les coordonnées
    public void setY(int p_Y) { ... }                  }
    public void set(Position p_Position) { ... }       }
    public void moveX(int p_dX) { ... }                } Incrémenter les coordonnées
    public void moveY(int p_dY) { ... }                }
  }
```

Notre modélisation est la suivante :

- Une classe de fourmis étendant la classe Thread et simulant le comportement d'une fourmi
- Une classe de groupe de fourmis étendant la classe ThreadGroup et représentant notre colonie
- Une classe environnement représentant le milieu dans lequel évoluent les fourmis (limites, graines)

A ces quatre classes (en comptant position) nous ajouterons la classe de la fenêtre graphique permettant de visualiser la simulation. Le résultat devrait ressembler à l'image ci à droite. Un cercle noir est une fourmi, un disque vert est une graine, un cercle noir sur un disque vert est une fourmis qui porte une graine ou qui marche sur une graine.



Nous allons commencer par créer chacune des classes avec un contenu minimum:

```
class Ant extends Thread
  {
    static int ReactivityDelay=10;
    static double ProbabilityToTakeFood=0.8;
    static double ProbabilityToPutDownFood=0.2;

    private boolean myCarryingFood = false;
    Environment myEnvironment;

    Position myPosition = new Position(0,0);
    boolean myAlive=true;

    public Ant(Environment p_Environment, String p_Name)
    {
      super(p_Environment.getAntGroup(),p_Name);
      myEnvironment = p_Environment;
    }

}
```

Temps de réaction moyen d'une fourmi

Probabilité de saisir une graine

Probabilité de reposer une graine

Booléen notant si la fourmi porte une graine

Référence de l'environnement de la fourmi

Position de la fourmi - (0,0) à sa naissance

Permet de savoir si la fourmi doit mourir

Constructeur

Les constantes de probabilité seront utilisées pour ajouter un peu d'aléatoire au comportement des fourmis. Elles sont statiques tout comme le temps de réaction d'une fourmi car elles sont communes à toutes les fourmis. L'aléatoire rend compte des multiples facteurs de perturbation qui ne sont pas simulés et permet aussi de sortir d'un minimum local comme dans les méthodes de recuit simulé ou comme la mutation dans les algorithmes génétiques. La composant aléatoire est souvent un facteur important dans les simulations à base d'agents réactifs car elle permet l'innovation. La référence vers l'objet `Environment` va permettre à la fourmi de percevoir l'environnement autour d'elle (présence de graines, limites du terrain) et d'agir sur lui (poser/prendre une graine)

```
class AntGroup extends ThreadGroup
  {
    public AntGroup(String p_Name) { super(p_Name); }
  }
```

Une colonie de fourmis est un groupe de Thead et porte un nom

```
class Environment extends Panel
  {
    final static int HEIGHT = 40;
    final static int WIDTH = 40;
    final static int UNIT = 10;

    int[][] myFoodGrid = new int [WIDTH][HEIGHT];

    AntGroup myAntGroup=null;

    public Environment()
    {
      myAntGroup= new AntGroup("AntGroup");
      setSize(WIDTH*UNIT,HEIGHT*UNIT);
    }
  }
```

L'environnement étend la classe Panel afin de pouvoir être affiché

L'environnement est un échiquier de 40x40

Une case de l'échiquier fait 10x10 unités graphiques

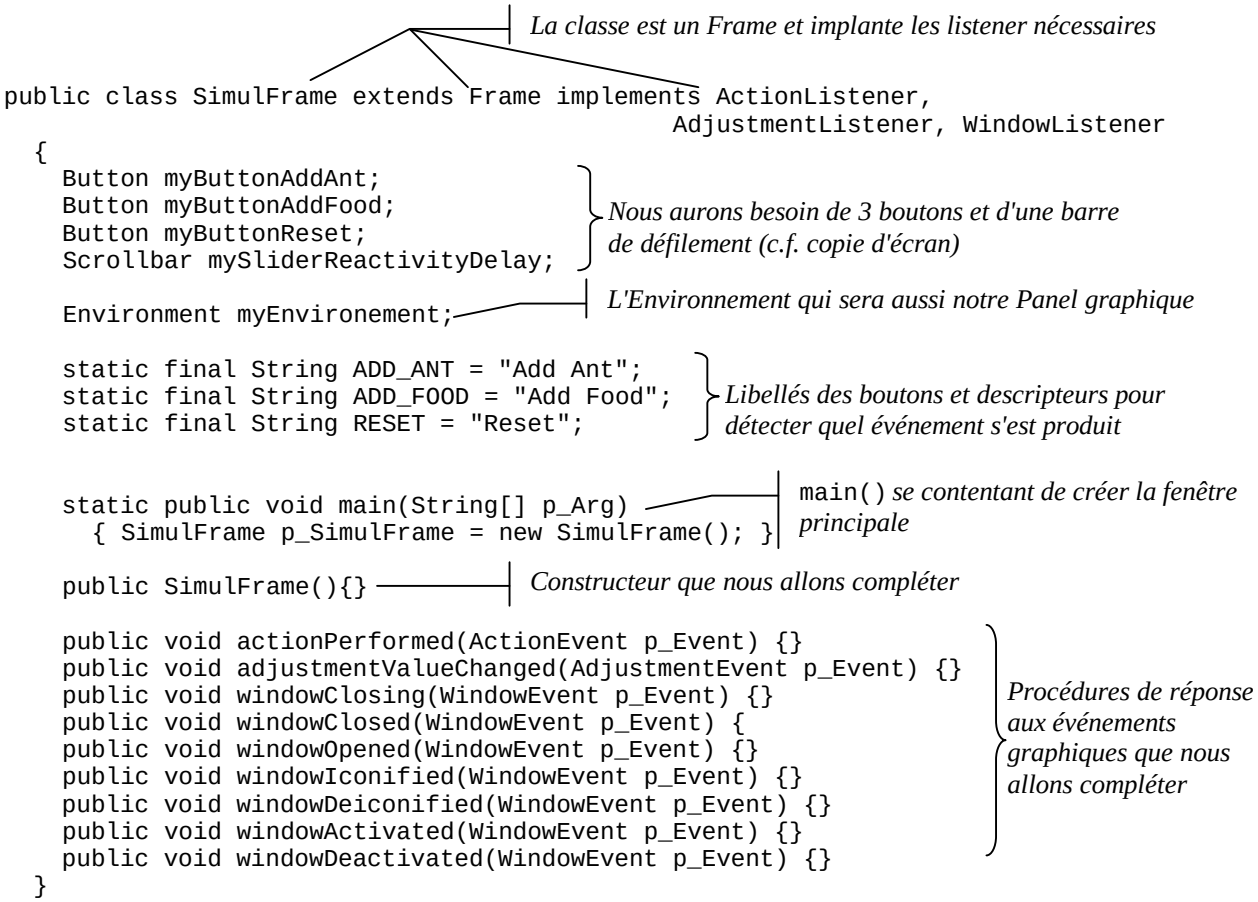
Ce tableau note le nombre de graines sur chaque case de l'échiquier

L'environnement a une référence sur la colonie de fourmis

Construction de l'environnement initial

Ces trois classes permettront donc de modéliser notre système réel. Nous allons les étendre petit à petit pour ajouter toutes les méthodes nécessaires.

Enfin voici le squelette de la classe principale qui sera aussi notre fenêtre d'affichage.




```
public class SimulFrame extends Frame implements ActionListener,
                                                  AdjustmentListener, WindowListener
  {
    Button myButtonAddAnt;
    Button myButtonAddFood;
    Button myButtonReset;
    Scrollbar mySliderReactivityDelay;

    Environment myEnvironement;

    static final String ADD_ANT = "Add Ant";
    static final String ADD_FOOD = "Add Food";
    static final String RESET = "Reset";


    static public void main(String[] p_Arg)
      { SimulFrame p_SimulFrame = new SimulFrame(); }

    public SimulFrame(){}

    public void actionPerformed(ActionEvent p_Event) {}
    public void adjustmentValueChanged(AdjustmentEvent p_Event) {}
    public void windowClosing(WindowEvent p_Event) {}
    public void windowClosed(WindowEvent p_Event) {
    public void windowOpened(WindowEvent p_Event) {}
    public void windowIconified(WindowEvent p_Event) {}
    public void windowDeiconified(WindowEvent p_Event) {}
    public void windowActivated(WindowEvent p_Event) {}
    public void windowDeactivated(WindowEvent p_Event) {}
  }
```

Exercice 5.12 Commencez par taper le squelette décrit précédemment. Vous devriez pouvoir compiler votre code, n'oubliez pas d'importer les librairie graphiques:

```
import java.awt.*;
import java.awt.event.ActionListener;
import java.awt.event.AdjustmentListener;
import java.awt.event.WindowListener;
import java.awt.event.ActionEvent;
import java.awt.event.AdjustmentEvent;
import java.awt.event.WindowEvent;
```

Ajoutez ensuite les méthodes suivantes:

Dans la classe `Ant` ajoutez la méthode `public void killMe()` qui suicide une fourmi et l'interrompt si elle dormait.

Dans la classe `AntGroup` ajoutez la méthode `public void killMe()` qui tue toutes les fourmis (pensez à regarder le chapitre sur les groupes de Thread)

Dans la classe Environment ajoutez les méthodes suivantes :

- `public void correct(Position p_Position)` qui corrige une position si elle sort des frontières du terrain, en la replaçant dans les frontières.
- `createAnt(String p_Name)` qui ajoute une fourmi à l'environnement et l'exécute.
- `public AntGroup getAntGroup()` qui renvoie la référence de la colonie de fourmis

Compilez et corrigez les éventuelles erreurs.

Exercice 5.13 On va maintenant s'intéresser à la classe fourmi :

1, programmez la procédure `public void moveRandomly()` qui déplace aléatoirement la fourmi sur l'une des 8 cases qui l'entourent et appelle la procédure de correction de l'environnement pour s'assurer qu'elle ne sort pas des limites.

2, programmez la procédure public `void paint(Graphics p_Graphics)` qui affiche sur l'objet graphique passé en argument, la fourmi comme un cercle noir si elle ne porte pas de graine et comme un cercle noir sur un disque vert si elle porte une graine. On utilisera les constantes de `Environment` : la position graphique utilisera HEIGHT et WIDTH et les dimensions du cercle et du disque seront égales à UNIT

3, Le comportement d'une fourmi est donné dans sa procédure principale:

```
public void run()
  {
    int l_foodQuantityAtMyPosition = 0;
    int l_foodQuantityArroundMyPosition = 0;

    while(myAlive)
      {
        l_foodQuantityAtMyPosition = myEnvironment.foodQuantityAt(myPosition);
        l_foodQuantityArroundMyPosition=
                                  myEnvironment.foodQuantityAround(myPosition);

      if(!myCarryingFood)
          {
            if (l_foodQuantityAtMyPosition>0)
              {
                if (Math.random()<
                        ProbabilityToTakeFood/l_foodQuantityArroundMyPosition)
                   myCarryingFood=myEnvironment.takeFoodAt(myPosition);
                else moveRandomly();
              }
            else moveRandomly();
          }
        else
          {
            if ((l_foodQuantityAtMyPosition==0) &&
                (Math.random()<
              (ProbabilityToPutDownFood*(1+l_foodQuantityArroundMyPosition))))
              {
                 myEnvironment.putDownFoodAt(myPosition);
                 myCarryingFood=false;
              }
            else moveRandomly();
          }
        myEnvironment.repaint();
        try
          {
            sleep((int)(ReactivityDelay*(Math.random()+0.5)));
          } catch (InterruptedException l_Exception) {}
      }
  }
```

La fourmi regarde si elle est sur une graine

La fourmi évalue la quantité de graine autour

Si la fourmi ne transporte pas de graine

Alors si elle est sur une graine elle la prend avec une probabilité inversement proportionnelle a la quantité environnante

Si elle ne la prend pas ou s'il n'y a pas de graine, elle bouge aléatoirement.

Si la fourmi transporte une graine

Alors si elle n'est pas sur une graine (pour éviter d'empiler les graines à l'écran) elle pose sa graine avec une probabilité proportionnelle a la quantité environnante

Si elle ne la pose pas elle bouge aléatoirement.

On rafraîchit l'affichage.

La fourmi s'endors pour une durée aléatoire centrée sur la valeur fixée (± 50%).

Le comportement des fourmis utilise des détections de l'environnement, qu'il va nous falloir programmer. Elle demande aussi à l'environnement de se rafraîchir graphiquement et il faudra gérer cela. Nous empêchons les fourmis d'empiler les graines les unes sur les autres pour que la visualisation soit meilleure. Vous noterez ici que le comportement de la fourmi est très basic et que pourtant le résultat émergeant de la colonie sera intelligent

Exercice 5.14 Il ne manque plus qu'une méthode à la classe `AntGroup`. Sur le même principe que la procédure `killMe()` programmez la méthode `void paint(Graphics p_Graphics)` qui appelle la procédure `paint(Graphics p_Graphics)` de chacune des fourmis du groupe en lui passant le paramètre graphique. Cette procédure permet de demander à toute la colonie de dessiner sa position, et permettra de rafraîchir l'affichage.

Exercice 5.15 La classe environnement demande un certain nombre de fonctions pour permettre les perceptions et les actions des fourmis :

1, Ajoutez la méthode suivante qui permettra d'ajouter aléatoirement de la nourriture dans l'environnement.

```
public void addFoodSomewhere()
    {
      myFoodGrid[(int)(Math.random()*WIDTH)][(int)(Math.random()*HEIGHT)]++;
      repaint();
    }
```

2, Programmez la procédure permettant à une fourmi de savoir la quantité de graines sur une case donnée:

```
int foodQuantityAt(Position p_Position)
```

Puis, programmez la procédure permettant à une fourmi de poser une graine sur une case donnée:

```
void putDownFoodAt(Position p_Position)
```

L'environnement est une ressource partagée, quelle est la conséquence sur ces deux méthodes ? qu'elle est le mot clef à ne pas oublier ici?

3, Voici la procédure permettant à une fourmi de prendre une graine, elle renvoie un booléen pour confirmer que tout s'est bien passé, car entre le moment où une fourmi détecte une graine et le moment où elle se décide à la prendre (après tirage aléatoire) une autre fourmi peut l'avoir prise...

```
public synchronized boolean takeFoodAt(Position p_Position)
    {
      int l_X = p_Position.getX();
      int l_Y = p_Position.getY();
      if (myFoodGrid[l_X][l_Y]>0)
        {
          myFoodGrid[l_X][l_Y]--;
          return(true);
        }
      else return(false);
    }
```

4, Programmez la méthode suivante qui pour une case donnée fait la somme des graines sur les huit cases environnante et la case concernée

```
int foodQuantityAround(Position p_Position)
```

Attention, n'oubliez pas le cas des cases aux limites du terrain. De plus on rappelle que l'environnement est une ressource partagée... n'oubliez pas le mot clef.

5, Pour remettre l'environnement à zéro on implante la procédure suivante :

```
public void reset()
    {
      myAntGroup.killMe();
      myFoodGrid = new int [WIDTH][HEIGHT];
      repaint();
    }
```

Elle tue les fourmis, efface les graines et rafraîchit l'affichage.

6, La partie assez complexe de l'affichage de l'environnement va vous être donnée, ajoutez-la au code:

Deux méthodes donnent la taille minimale du Panel pour l'affichage. Si on ne faisait pas cela, la fenêtre pourrait être trop réduite car le système ne sait pas quelle est la taille nécessaire pour afficher notre fourmilière...

```
public Dimension getMinimumSize() { return new Dimension(WIDTH*UNIT,HEIGHT*UNIT);}

public Dimension getPreferredSize() {return getMinimumSize();}
```

Ajoutez ces champs: ils permettent de créer une image de l'environnement et de l'afficher pendant que l'on dessine la suivante, l'affichage est alors sensiblement amélioré.

```
Image myEnvironmentImage = null;
Dimension myEnvironmentImageSize = null;
Graphics myEnvironmentGraphics = null;
```

Enfin ajoutez la méthode de dessin de l'environnement, elle surcharge la méthode `paint(Graphics p_Graphics)` héritée de Panel :

```
public void paint(Graphics p_Graphics)
    {
      if (myEnvironmentImage == null)
        {
          myEnvironmentImageSize = getSize();
          myEnvironmentImage = createImage((int)myEnvironmentImageSize.getWidth(),
                                           (int)myEnvironmentImageSize.getHeight());
        }
      p_Graphics.drawImage(myEnvironmentImage, 0, 0, null);
```

Afficher l'ancienne image en attendant la nouvelle.

```
      myEnvironmentGraphics = myEnvironmentImage.getGraphics();
      myEnvironmentGraphics.setColor(getBackground());
      myEnvironmentGraphics.fillRect(0, 0, (int)myEnvironmentImageSize.getWidth(),
                                           (int)myEnvironmentImageSize.getHeight());
```

Faire un fond blanc.

```
      myEnvironmentGraphics.setColor(Color.green);
      for(int l_X=0; l_X < WIDTH ; l_X++)
        for(int l_Y=0; l_Y < HEIGHT; l_Y++)
          if (myFoodGrid[l_X][l_Y]>0)
            myEnvironmentGraphics.fillOval(l_X*UNIT,l_Y*UNIT,UNIT,UNIT);
```

Dessiner les graines vertes

```
      myAntGroup.paint(myEnvironmentGraphics);
```

Demander aux fourmis de s'afficher

```
      p_Graphics.drawImage(myEnvironmentImage, 0, 0, null);
    }
```

Afficher la nouvelle image

Nous ne rentrerons pas dans les détails de cette méthode graphique, elle a simplement l'avantage de réduire le clignotement de l'affichage en maintenant l'ancienne image à l'écran pendant que la nouvelle est calculée.

A ce stade vous devriez pouvoir compiler le programme. Faites le et faites la chasse aux erreurs. Ne le lancez pas il n'y a rien à voir pour le moment.

Exercice 5.16 La classe principale demande un travail graphique pour créer l'interface (cf. image d'écran donnée en introduction)

1, Dans le constructeur, utilisez ce que vous avez appris sur AWT pour:

- Appeler le constructeur de la classe mère (Frame) et donner le titre "Ant sort simulation"
- Préparer un Layout du type BorderLayout
- ajouter un WidowListener qui sera la classe elle-même (addWindowListener(this);)
- initialiser la variable myEnvironement avec une nouvelle instance de la classe Environment et l'ajouter à la fenêtre
- Créer un Panel avec un fond gris, instancier et lui ajouter les trois boutons définis comme champs de la classe et dont le titre et l'action sont aussi donnés par des champs de la classe. Pensez à mettre la classe principale comme listeners de ces boutons.
- De même, instanciez et ajoutez au Panel une barre de défilement elle aussi définie comme champ de la classe : mySliderReactivityDelay = new Scrollbar(Scrollbar.HORIZONTAL, 10, 100, 1, 1100); Là encore la classe principale sera donnée comme listener de la barre.
- Ajoutez le Panel à la fenêtre, faites en sorte que la fenêtre prenne sa taille minimale, s'affiche et ne puisse plus être redimensionnée

2, On modifie maintenant trois méthodes jusqu'à maintenant vide:

```
public void actionPerformed(ActionEvent p_Event)
  {
    String l_Command = p_Event.getActionCommand();
    if (l_Command.equals(ADD_ANT))
      {
        myEnvironement.createAnt("AnAnt");
      }
    else if (l_Command.equals(ADD_FOOD))
      {
        for (int l_Meter=1;l_Meter<5;l_Meter++) myEnvironement.addFoodSomewhere();
      }
    else if (l_Command.equals(RESET))
      {
        myEnvironement.reset();
      }
  }
```

L'action du bouton ADD_ANT est d'ajouter une fourmi nommée "AnAnt"

L'action du bouton ADD_FOOD *est d'ajouter 5 graines*

L'action du RESET est de réinitialiser l'environnement

La barre de défilement modifie le temps moyen de réaction des fourmis

```
public void adjustmentValueChanged(AdjustmentEvent p_Event)
  { Ant.ReactivityDelay=mySliderReactivityDelay.getValue(); }

public void windowClosing(WindowEvent p_Event)
  { myEnvironement.reset(); dispose(); System.exit(0); }
```

La fenêtre se fermant, il faut détruire l'environnement, la fenêtre et sortir de l'application.

Vous pouvez maintenant compiler et lancer le projet. Ajoutez de la nourriture en appuyant une dizaine de fois sur "Add Food", puis Ajoutez des fourmis en appuyant une dizaine de fois sur "Add Ant". N'hésitez pas à ajouter des fourmis et de la nourriture. Utilisez la barre de défilement pour régler la vitesse du système.

Références :

www.javasoft.com
http://raphaello.univ-fcomte.fr/Java/
http://perso.wanadoo.fr/guillaume/