1. Le multithreading

1.1 Multitâche et thread  
1.2 Java autorise l'utilisation des threads  
1.3 L'interface Runnable  
1.4 La classe java.lang.Thread  
1.5 Méthodes de base sur les objets  de Thread

2. Exercice entièrement traité

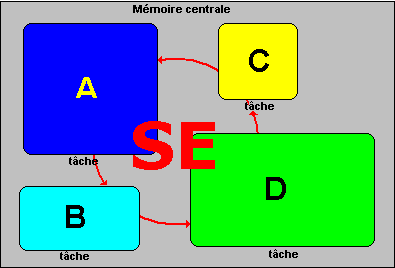
2.1 Enoncé et classes  
2.2 première solution sans threads    
2.3 deuxième solution avec des threads  
2.4 variation sur les threads

1. Le Multithreading

Nous savons que les ordinateurs fondés sur les principes d'une machine de Von Neumann, sont des machines séquentielles donc n'exécutant qu'une seule tâche à la fois. Toutefois, le gaspillage de temps engendré par cette manière d'utiliser un ordinateur (le processeur central passe l'écrasante majorité de son temps à attendre) a très vite été endigué par l'invention de systèmes d'exploitations de multi-programmation ou multi-tâches, permettant l'exécution "simultanée" de plusieurs tâches. Dans un tel système, les différentes tâches sont exécutées sur une machine disposant d'un seul processeur, en apparence en même temps ou encore en parallèle, en réalité elles sont exécutées séquentiellement chacune à leur tour, ceci ayant lieu tellement vite pour notre conscience que nous avons l'impression que les programmes s'exécutent simultanément. Rappelons ici qu'une tâche est une application comme un traitement de texte, un navigateur internet, un jeu,... ou d'autres programmes spécifiques au système d'exploitation que celui-ci exécute.

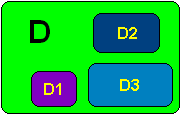
1.1 Multitâche et thread

Le noyau du système d'exploitation SE, conserve en permanence le contrôle du temps d'exécution en distribuant cycliquement des tranches de temps (time-slicing) à chacune des applications  A, B, C et D figurées ci-dessous. Dans cette éventualité, une application représente dans le système un processus :

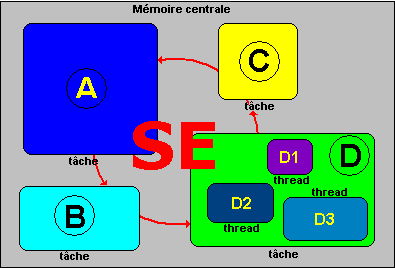


Rappelons la définition donnée par A.Tanenbaum des processus : un programme qui s'exécute et qui possède son propre espace mémoire : ses registres, ses piles, ses variables et son propre processeur virtuel (simulé en multi-programmation par la commutation entre processus effectuée par le processeur unique).

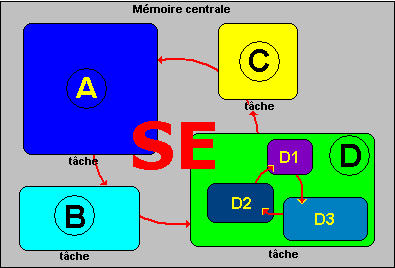
Thread  
En fait, chaque processus peut lui-même fonctionner comme le système d'exploitation en lançant des sous-tâches internes au processus et par là même reproduire le fonctionnement de la  multi-programmation. Ces sous-tâches sont nommées "flux d'exécution" ou Threads. Ci-dessous nous supposons que l'application D exécute en même temps les 3 Threads D1, D2 et D3 :



Reprenons l'exemple d'exécution précédent, dans lequel 4 processus s'exécutent "en même temps" et incluons notre processus D possédant 3 flux d'exécutions (threads)  :



La commutation entre les threads d'un processus  fonctionne de le même façon que la commutation entre les processus, chaque thread se voit alloué cycliquement, lorsque le processus D est exécuté une petite tranche de temps. Le partage et la répartition du temps sont effectués uniquement par le système d’exploitation :



Multithreading et processus  
  
Définition  :

La majorité des systèmes d'exploitation (Windows, Solaris, MacOs,...) supportent l'utilisation d'application contenant des threads, l'on désigne cette fonctionnalité sous le nom de Multithreading.

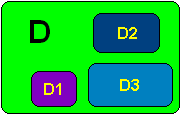
Différences entre threads et processus :

Communication entre threads plus rapide que la communication entre processus,

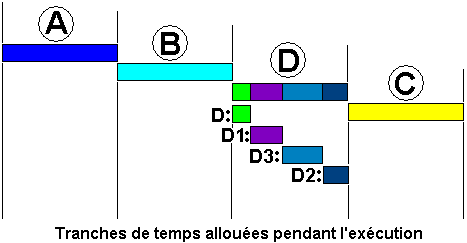
Les threads partagent un même espace de mémoire (de travail) entre eux,

Les processus ont chacun un espace mémoire personnel.

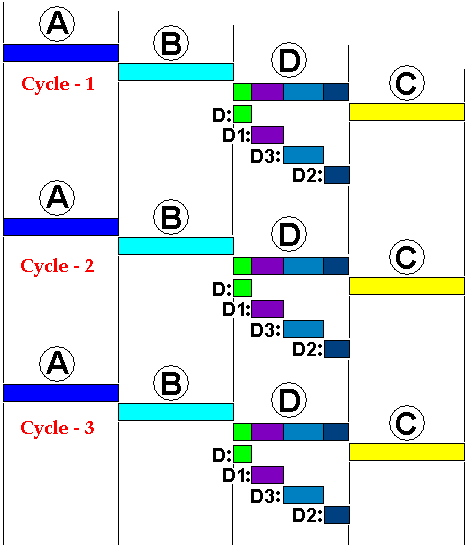
Dans l'exemple précédent, figurons les processus A, B, C et  le processus D avec ses threads dans un graphique représentant une tranche de temps d'exécution allouée par le système et supposée être la même pour chaque processus.



Le système ayant alloué le même temps d"exécution à chaque processus, lorsque par exemple le tour vient au processus D de s'exécuter dans sa tranche de temps, il exécutera une petite sous-tranche pour D1, pour D2, pour D3 et attendra le prochain cycle. Ci-dessous un cycle d'exécution :

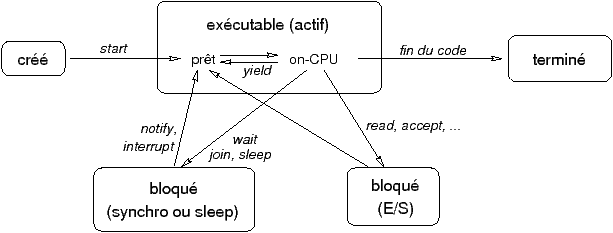


Voici sous les mêmes hypothèses de temps égal d'exécution alloué à chaque processus, le comportement de l'exécution sur 3 cycles consécutifs :



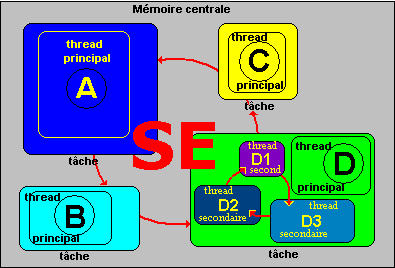
Les langages Delphi, Java et C# disposent  chacun de classes permettant d'écrire et d'utiliser des threads dans vos applications.

Le cycle de vie d’une activité Java est similaire au cycle de vie standard d’une activité des Threads Posix, ou d’un processus :



1.2 Java autorise l'utilisation des threads

Lorsqu'un programme Java s'exécute en dehors d'une programmation de multi-threading, le processus associé comporte automatiquement un thread appelé thread principal. Un autre thread utilisé dans une application s'appelle un thread secondaire. Supposons que les quatre applications (ou tâches) précédentes A, B, C et  D soient toutes des applications Java, et que D soit celle qui comporte trois threads secondaires D1, D2 et D3 "parallèlement" exécutés :



En Java c'est  l'interface Runnable qui permet l'utilisation des threads dans la Java machine. Voici une déclaration de cette interface :

|  |
| --- |
| public interface Runnable {    public void run( ) ; } |

Cette interface Runnable doit obligatoirement être implémentée par toute classe dont les instances seront exécutées par un thread. Une telle classe implémentant l'interface Runnable doit alors implémenter la méthode abstraite sans paramètre  public void run( ) . Cette interface est conçue pour mettre en place un protocole commun à tous les objets qui souhaitent exécuter du code pendant que eux-mêmes sont actifs.

L'interface Runnable est essentiellement implantée par  la classe Thread du package java.lang :

java.lang.Object  
  |  
 +--java.lang.Thread

Nous voyons maintenant comment concevoir une classe personnalisée dénommée MaClasse qui permettre l'exécution de ses instances dans des threads machines. Java vous offre deux façons différentes de décrire la classe MaClasse utilisant des threads  :

Soit par implémentation de l'interface Runnable.

Soit par héritage de la classe java.lang.Thread

Interface Runnable  
On effectue ce choix habituellemnt lorsque l'on veut utiliser une classe ClassB héritant d'une classe ClassA et  que cette ClassB fonctionne comme un thread : il faudrait donc pouvoir hériter à la fois de la classe ClassA et de la classe Thread, ce qui est impossible, java ne connaît pas l'héritage multiple. En implémentant l'interface Runnable dans la  classe ClassB, on rajoute à notre classe ClassB une nouvelle méthode run ( ) (qui est en fait la seule méthode de l'interface Runnable).

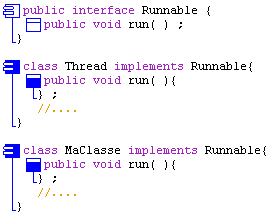
Classe Thread  
Dans le second cas l'héritage à partir de la classe Thread qui implémente l'interface Runnable et qui permet l'utilisation de méthodes de manipulation d'un thread (mise en attente, reprise,...).

|  |
| --- |
| Dans les deux cas, il vous faudra instancier un objet de classe Thread et vous devrez  mettre  ou invoquer le code à "exécuter en parallèle" dans le corps de la méthode run ( ) de votre classe . |

1.3 L'interface Runnable

Première version Maclasse implémente l'interface Runnable :  
vous héritez d'une seule méthode run ( ) que vous devez implémenter

Une classe dénommée MaClasse :



Exemple de modèle :  
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres de 1 à 100

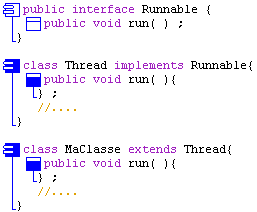
|  |
| --- |
| public class MaClasse implements Runnable {   Thread  UnThread ;     MaClasse ( ) {    //....initialisations éventuelles du constructeur de MaClasse    UnThread = new Thread ( this , "thread secondaire" );    UnThread.start( ) ;     }     public void run ( ) {    //....actions du thread secondaire ici    for ( int i1=1; i1<100; i1++)             System.out.println(">>> i1 = "+i1);    } } |

Pour utiliser cette classe contenant un thread secondaire, il faut instancier un objet de cette classe :

|  |
| --- |
| class UtiliseMaclasse {   public static void main(String[] x) {      MaClasse tache1 = new MaClasse (  );      /\*.......le thread secondaire a été créé et activé      le reste du code concerne le thread principal \*/      for ( int i2 =1; i2<100;i2++)             System.out.println(" i2 = "+i2);   } } |

1.4 La classe java.lang.Thread

Deuxième version Maclasse dérive de la classe Thread  :  
vous héritez de toutes les méthodes de la classe Thread  
dont la méthode run( ) que vous devez redéfinir



Exemple de modèle :  
les 2 threads principal et secondaire affichent chacun "en même temps" une liste de nombres de 1 à 100

|  |
| --- |
| public class MaClasse extends Thread {     MaClasse ( ) {      //....initialisations éventuelles du constructeur de MaClasse      this .start( ) ;     }     public void run ( ) {    //....actions du thread secondaire ici    for ( int i1=1; i1<100; i1++)             System.out.println(">>> i1 = "+i1);    } } |

Lorsque vous voulez utiliser cette classe contenant un thread secondaire, il vous faudra instancier un objet de cette classe de la même manière que pour la première version :

|  |
| --- |
| class UtiliseMaclasse {   public static void main(String[] x) {      MaClasse tache1 = new MaClasse (  );      /\*.......le thread secondaire a été créé et activé     le reste du code concerne le thread principal \*/     for ( int i2 =1; i2<100;i2++)             System.out.println(" i2 = "+i2);   } } |

Les deux versions par interface Runnable ou classe Thread, produisent le même résultat d'exécution. Chaque thread affiche séquentiellement sur la console ses données lorsque le système lui donne la main, ce qui donne un "mélange des deux affichages" :  
  
Résultats d'exécution :

|  |  |  |  |
| --- | --- | --- | --- |
| i2 = 1 i2 = 2 i2 = 3 i2 = 4 i2 = 5 i2 = 6 i2 = 7 i2 = 8 i2 = 9 i2 = 10 i2 = 11 i2 = 12 i2 = 13 i2 = 14 i2 = 15 i2 = 16 i2 = 17 i2 = 18 i2 = 19 i2 = 20 i2 = 21 i2 = 22 i2 = 23 i2 = 24 i2 = 25 i2 = 26 i2 = 27 i2 = 28 i2 = 29 i2 = 30 i2 = 31 i2 = 32 i2 = 33 i2 = 34 i2 = 35 i2 = 36 i2 = 37 i2 = 38 i2 = 39 i2 = 40 i2 = 41 i2 = 42 i2 = 43 i2 = 44 i2 = 45 i2 = 46 i2 = 47 i2 = 48 i2 = 49 | i2 = 50 i2 = 51 i2 = 52 i2 = 53 i2 = 54 i2 = 55 i2 = 56 i2 = 57 i2 = 58 >>> i1 = 1 i2 = 59 >>> i1 = 2 i2 = 60 >>> i1 = 3 i2 = 61 >>> i1 = 4 i2 = 62 i2 = 63 i2 = 64 i2 = 65 i2 = 66 i2 = 67 i2 = 68 i2 = 69 i2 = 70 i2 = 71 i2 = 72 i2 = 73 i2 = 74 i2 = 75 i2 = 76 i2 = 77 i2 = 78 i2 = 79 i2 = 80 i2 = 81 i2 = 82 i2 = 83 i2 = 84 i2 = 85 i2 = 86 i2 = 87 i2 = 88 i2 = 89 i2 = 90 i2 = 91 i2 = 92 i2 = 93 i2 = 94 | i2 = 95 i2 = 96 i2 = 97 i2 = 98 i2 = 99 >>> i1 = 5 >>> i1 = 6 >>> i1 = 7 >>> i1 = 8 >>> i1 = 9 >>> i1 = 10 >>> i1 = 11 >>> i1 = 12 >>> i1 = 13 >>> i1 = 14 >>> i1 = 15 >>> i1 = 16 >>> i1 = 17 >>> i1 = 18 >>> i1 = 19 >>> i1 = 20 >>> i1 = 21 >>> i1 = 22 >>> i1 = 23 >>> i1 = 24 >>> i1 = 25 >>> i1 = 26 >>> i1 = 27 >>> i1 = 28 >>> i1 = 29 >>> i1 = 30 >>> i1 = 31 >>> i1 = 32 >>> i1 = 33 >>> i1 = 34 >>> i1 = 35 >>> i1 = 36 >>> i1 = 37 >>> i1 = 38 >>> i1 = 39 >>> i1 = 40 >>> i1 = 41 >>> i1 = 42 >>> i1 = 43 >>> i1 = 44 >>> i1 = 45 >>> i1 = 46 >>> i1 = 47 >>> i1 = 48 >>> i1 = 49 | >>> i1 = 50 >>> i1 = 51 >>> i1 = 52 >>> i1 = 53 >>> i1 = 54 >>> i1 = 55 >>> i1 = 56 >>> i1 = 57 >>> i1 = 58 >>> i1 = 59 >>> i1 = 60 >>> i1 = 61 >>> i1 = 62 >>> i1 = 63 >>> i1 = 64 >>> i1 = 65 >>> i1 = 66 >>> i1 = 67 >>> i1 = 68 >>> i1 = 69 >>> i1 = 70 >>> i1 = 71 >>> i1 = 72 >>> i1 = 73 >>> i1 = 74 >>> i1 = 75 >>> i1 = 76 >>> i1 = 77 >>> i1 = 78 >>> i1 = 79 >>> i1 = 80 >>> i1 = 81 >>> i1 = 82 >>> i1 = 83 >>> i1 = 84 >>> i1 = 85 >>> i1 = 86 >>> i1 = 87 >>> i1 = 88 >>> i1 = 89 >>> i1 = 90 >>> i1 = 91 >>> i1 = 92 >>> i1 = 93 >>> i1 = 94 >>> i1 = 95 >>> i1 = 96 >>> i1 = 97 >>> i1 = 98 >>> i1 = 99 |

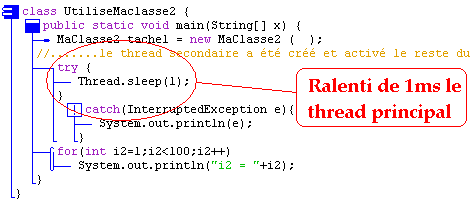
 ---- operation complete.

1.5 Méthodes de base sur les objets  de la classe Thread  
  
  
Les méthodes static de la classe Thread travaillent sur le thread courant  
(en l'occurrence le thread qui invoque la méthode static) :

|  |  |
| --- | --- |
| static int activeCount( ) | renvoie le nombre total de threads actifs actuellement. |
| static Thread currentThread( ) | renvoie la référence de l'objet thread actuellement exécuté (thread courant) |
| static void dumpStack( ) | Pour le debugging : une trace de la pile du thread courant |
| static int enumerate( Thread [] tarray ) | Renvoie dans le tableau tarray[] les reférences de tous les threads actifs actullement. |
| static boolean    interrupted( ) | Teste l'indicateur d'interruption du thread courant, mais change cet interrupteur (utiliser plutôt la méthode d'instance boolean isInterrupted( ) ). |
| static void    sleep( long millis, int nanos ) | Provoque l'arrêt de l'exécution du thread courant pendant milis  milisecondes et nanos nanosecondes. |
| static void    sleep( long millis ) | Provoque l'arrêt de l'exécution du thread courant pendant milis  milisecondes. |

Reprenons l'exemple ci-haut de l'affichage entre-mêlé de deux listes d'entiers de 1 à 100 :

|  |
| --- |
| class MaClasse2 extends Thread {     MaClasse2 ( ) {      this .start( ) ;     }     public void run ( ) {    for ( int i1=1; i1<100; i1++)             System.out.println(">>> i1 = "+i1);    } }  class UtiliseMaclasse2 {   public static void main(String[] x) {      // corps du thread principal      MaClasse2 tache1 = new MaClasse2 (  ); // le thread secondaire      for ( int i2 =1; i2<100;i2++)             System.out.println(" i2 = "+i2);   } } |

Arrêtons pendant une milliseconde l'exécution du thread principal : Thread.sleep(1);  
comme la méthode de classe sleep( ) propage une InterruptedException nous devons l'intercepter :  
  
  
  
En arrêtant pendant 1ms le thread principal, nous laissons donc plus de temps au thread secondaire pour s'exécuter, les affichages ci-dessous le montrent :  
  
Résultats d'exécution :

|  |  |  |  |
| --- | --- | --- | --- |
| >>> i1 = 1 >>> i1 = 2 >>> i1 = 3 >>> i1 = 4 >>> i1 = 5 >>> i1 = 6 >>> i1 = 7 >>> i1 = 8 >>> i1 = 9 >>> i1 = 10 >>> i1 = 11 >>> i1 = 12 >>> i1 = 13 >>> i1 = 14 >>> i1 = 15 >>> i1 = 16 >>> i1 = 17 >>> i1 = 18 >>> i1 = 19 >>> i1 = 20 >>> i1 = 21 >>> i1 = 22 >>> i1 = 23 >>> i1 = 24 >>> i1 = 25 >>> i1 = 26 >>> i1 = 27 >>> i1 = 28 >>> i1 = 29 >>> i1 = 30 >>> i1 = 31 >>> i1 = 32 >>> i1 = 33 >>> i1 = 34 >>> i1 = 35 >>> i1 = 36 >>> i1 = 37 >>> i1 = 38 >>> i1 = 39 >>> i1 = 40 >>> i1 = 41 >>> i1 = 42 >>> i1 = 43 >>> i1 = 44 >>> i1 = 45 >>> i1 = 46 >>> i1 = 47 >>> i1 = 48 >>> i1 = 49 | >>> i1 = 50 >>> i1 = 51 >>> i1 = 52 >>> i1 = 53 >>> i1 = 54 >>> i1 = 55 >>> i1 = 56 >>> i1 = 57 >>> i1 = 58 >>> i1 = 59 >>> i1 = 60 >>> i1 = 61 >>> i1 = 62 >>> i1 = 63 >>> i1 = 64 >>> i1 = 65 >>> i1 = 66 >>> i1 = 67 >>> i1 = 68 >>> i1 = 69 >>> i1 = 70 >>> i1 = 71 >>> i1 = 72 >>> i1 = 73 >>> i1 = 74 >>> i1 = 75 >>> i1 = 76 >>> i1 = 77 >>> i1 = 78 >>> i1 = 79 >>> i1 = 80 >>> i1 = 81 i2 = 1 >>> i1 = 82 i2 = 2 >>> i1 = 83 i2 = 3 >>> i1 = 84 i2 = 4 >>> i1 = 85 i2 = 5 >>> i1 = 86 i2 = 6 i2 = 7 i2 = 8 i2 = 9 i2 = 10 i2 = 11 i2 = 12 | i2 = 13 i2 = 14 i2 = 15 i2 = 16 i2 = 17 i2 = 18 i2 = 19 i2 = 20 i2 = 21 i2 = 22 i2 = 23 i2 = 24 i2 = 25 i2 = 26 i2 = 27 i2 = 28 i2 = 29 i2 = 30 i2 = 31 i2 = 32 i2 = 33 i2 = 34 i2 = 35 i2 = 36 i2 = 37 i2 = 38 i2 = 39 i2 = 40 i2 = 41 i2 = 42 i2 = 43 i2 = 44 i2 = 45 i2 = 46 i2 = 47 i2 = 48 i2 = 49 i2 = 50 i2 = 51 >>> i1 = 87 i2 = 52 >>> i1 = 88 i2 = 53 >>> i1 = 89 i2 = 54 >>> i1 = 90 >>> i1 = 91 >>> i1 = 92 >>> i1 = 93 >>> i1 = 94 | >>> i1 = 95 >>> i1 = 96 >>> i1 = 97 >>> i1 = 98 >>> i1 = 99 i2 = 55 i2 = 56 i2 = 57 i2 = 58 i2 = 59 i2 = 60 i2 = 61 i2 = 62 i2 = 63 i2 = 64 i2 = 65 i2 = 66 i2 = 67 i2 = 68 i2 = 69 i2 = 70 i2 = 71 i2 = 72 i2 = 73 i2 = 74 i2 = 75 i2 = 76 i2 = 77 i2 = 78 i2 = 79 i2 = 80 i2 = 81 i2 = 82 i2 = 83 i2 = 84 i2 = 85 i2 = 86 i2 = 87 i2 = 88 i2 = 89 i2 = 90 i2 = 91 i2 = 92 i2 = 93 i2 = 94 i2 = 95 i2 = 96 i2 = 97 i2 = 98 i2 = 99 |

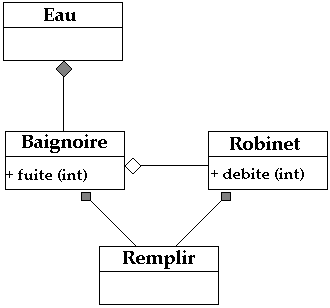
 ---- operation complete.  
  
Les méthodes d'instances de la classe Thread  
  
Java permet d'arrêter (stop), de faire attendre (sleep, yield), d'interrompre (interrupt), de changer la priorité(setPriority), de détruire(destroy) des threads à travers les méthodes de la classe Thread.  
  
Java permet à des threads de "communiquer" entre eux sur leur état (join, notify, wait)

2. Exercice entièrement traité

2.1 Enoncé et classes

Nous vous proposons de programmer une simulation du problème qui  a tant mis à contribution les cerveaux des petits écoliers d'antan : le problème du robinet qui remplit d'eau une baignoire qui fuit. Nous allons écrire un programme qui simulera le remplissage de la baignoire par un robinet dont le débit est connu et paramètrable. La baignoire a une fuite dont le débit lui aussi est connu et paramétrable. Dès que la baignoire est entièrement vide l'on colmate la fuite et tout rentre dans l'ordre. On arrête le programme dès que la baignoire est pleine que la fuite soit colmatée ou non.

Nous choisissons le modèle objet pour représenter notre problème :

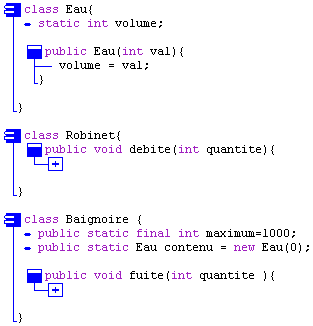


Une classe Eau qui contient un champ static indiquant le volume d'eau actuel de l'objet auquel il appartient.

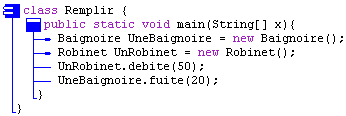
Une classe Baignoire possédant un contenu (en litres d'eau) et une fuite qui diminue le volume d'eau du contenu de la baignoire.

Une classe Robinet qui débite (augmente) le volume d'eau du contenu de la baignoire d'une quantité fixée.

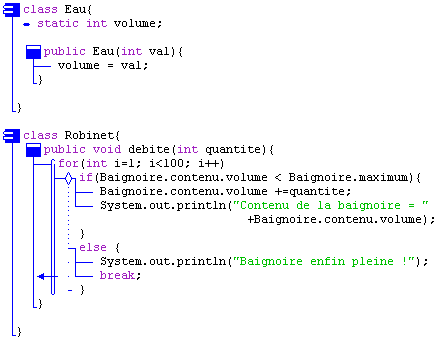
2.2 Une première solution sans les threads

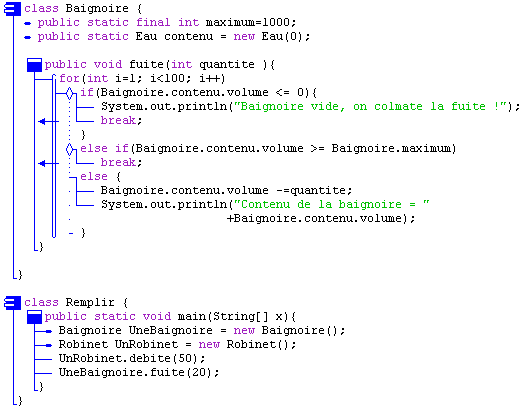


Une classe Remplir qui permet le démarrage des actions : mise en place de la baignoire, du robinet, ouverture du robinet et fuite de la baignoire :



Nous programmons les méthodes debite( int quantite) et fuite( int quantite) de telle sorte qu'elles afichent chacune l'état du contenu de la baignoire après que l'apport ou la diminution d'eau a eu lieu. Nous simulerons et afficherons pour chacune des deux méthodes 100 actions de base (100 diminutions pour la méthode fuite et 100 augmentations pour la méthode debite).





Résultats d'exécution :  
Contenu de la baignoire = 50  
Contenu de la baignoire = 100  
Contenu de la baignoire = 150  
Contenu de la baignoire = 200  
Contenu de la baignoire = 250  
Contenu de la baignoire = 300  
Contenu de la baignoire = 350  
Contenu de la baignoire = 400  
Contenu de la baignoire = 450  
Contenu de la baignoire = 500  
Contenu de la baignoire = 550  
Contenu de la baignoire = 600  
Contenu de la baignoire = 650  
Contenu de la baignoire = 700  
Contenu de la baignoire = 750  
Contenu de la baignoire = 800  
Contenu de la baignoire = 850  
Contenu de la baignoire = 900  
Contenu de la baignoire = 950  
Contenu de la baignoire = 1000  
Baignoire enfin pleine !

 ---- operation complete.

Que s'est-il passé ?

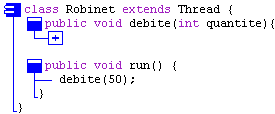
La programmation séquentielle du problème n'a pas permis d'exécuter l'action de fuite de la baignoire puisque nous avons arrêté le processus dès que la baignoire était pleine. En outre nous n'avons pas pu simuler le remplissage et le vidage "simultanés" de la baignoire.

Nous allons utiliser deux threads (secondaires) pour rendre la simulation plus réaliste, en essayant de faire les actions de débit-augmentation et fuite-diminution en parallèle.

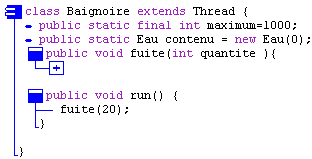
2.3 Deuxième solution avec des threads

Les objets qui produisent les variations du volume d'eau sont le robinet et la baignoire, ce sont eux qui doivent être "multi-threadés".

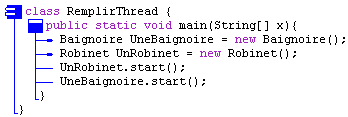
Reprenons pour cela la classe Robinet en la dérivant de la classe Thread, et en redéfinissant la méthode run( ) qui doit contenir le code de débit à exécuter en "parallèle" (le corps de la méthode debite n'a pas changé) :



De même en dérivant la classe Baignoire de la classe Thread, et en redéfinissant la méthode run( ) avec le code de fuite à exécuter en "parallèle" (le corps de la méthode fuite n'a pas changé) :



Enfin la classe RemplirThread qui permet le démarrage des actions : mise en place de la baignoire, du robinet, puis lancement en parallèle de l'ouverture du robinet et de la fuite de la baignoire :



Résultats d'exécution obtenus :

|  |  |
| --- | --- |
| Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250 Remplissage, contenu de la baignoire = 300 Remplissage, contenu de la baignoire = 350 Remplissage, contenu de la baignoire = 400 Remplissage, contenu de la baignoire = 450 Remplissage, contenu de la baignoire = 500 Remplissage, contenu de la baignoire = 550 Remplissage, contenu de la baignoire = 600 Fuite, contenu de la baignoire = 580 Remplissage, contenu de la baignoire = 630 Fuite, contenu de la baignoire = 610 Remplissage, contenu de la baignoire = 660 Fuite, contenu de la baignoire = 640 Remplissage, contenu de la baignoire = 690 Fuite, contenu de la baignoire = 670 Remplissage, contenu de la baignoire = 720 Remplissage, contenu de la baignoire = 770 Remplissage, contenu de la baignoire = 820 Remplissage, contenu de la baignoire = 870 Fuite, contenu de la baignoire = 850 | Remplissage, contenu de la baignoire = 900 Fuite, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Fuite, contenu de la baignoire = 910 Fuite, contenu de la baignoire = 890 Fuite, contenu de la baignoire = 870 Fuite, contenu de la baignoire = 850 Fuite, contenu de la baignoire = 830 Fuite, contenu de la baignoire = 810 Fuite, contenu de la baignoire = 790 Fuite, contenu de la baignoire = 770 Fuite, contenu de la baignoire = 750 Fuite, contenu de la baignoire = 730 Fuite, contenu de la baignoire = 710 Fuite, contenu de la baignoire = 690 Remplissage, contenu de la baignoire = 740 Fuite, contenu de la baignoire = 720 Remplissage, contenu de la baignoire = 770 Remplissage, contenu de la baignoire = 820 Remplissage, contenu de la baignoire = 870 Remplissage, contenu de la baignoire = 920 Remplissage, contenu de la baignoire = 970 Remplissage, contenu de la baignoire = 1020 Baignoire enfin pleine ! |

 ---- operation complete.  
  
Nous voyons que les deux threads s'exécutent cycliquement (mais pas d'une manière égale) selon un ordre non déterministe sur lequel nous n'avons pas de prise mais qui dépend de la java machine et du système d'exploitation, ce qui donnera des résultats différents à chaque nouvelle exécution. Le paragraphe suivant montre un exemple ou nous pouvons contraindre des threads de "dialoguer" pour laisser la place l'un à l'autre

2.4 variation sur les threads

Lorsque la solution adoptée est l'héritage à partir de la classe Thread, vous pouvez agir sur l'ordonnancement d'exécution des threads présents. Dans notre exemple utilisons deux méthodes de cette classe Thread :

|  |
| --- |
| void setPriority ( int newPriority)  static void yield ( ) |

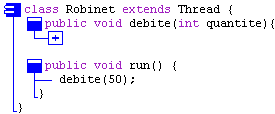
Privilégions le thread Robinet grâce à la méthode setPriority  :

La classe Thread possède 3 champs static permettant d'attribuer 3 valeurs de priorités différentes, de la plus haute à la plus basse, à un thread indépendamment de l'échelle réelle du système d'exploitation sur lequelle travaille la Java Machine :

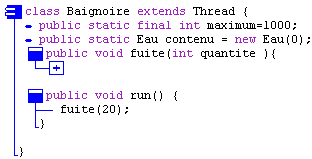
|  |  |
| --- | --- |
| static int    MAX\_PRIORITY | La priorité maximum que peut avoir un thread. |
| static int    MIN\_PRIORITY | La priorité minimum que peut avoir un thread. |
| static int    NORM\_PRIORITY | La priorité par défaut attribuée à un thread. |

La méthode setPriority appliquée à une instance de thread change sa priorité d'exécution. Nous mettons l'instance UnRobinet à la priorité maximum setPriority(Thread.MAX\_PRIORITY) :

Classe Robinet sans changement



Classe Baignoire sans changement



Voici le changement de code dans la classe principale :  
  
class RemplirThread {  
       public static void main(String[] x){  
         Baignoire UneBaignoire = new Baignoire();  
         Robinet UnRobinet = new Robinet( );  
         UnRobinet.setPriority ( Thread.MAX\_PRIORITY );  
         UnRobinet.start( );  
         UneBaignoire.start( );  
      }  
 }

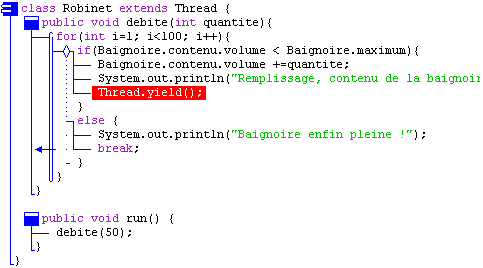
Résultats d'exécution obtenus :

|  |  |
| --- | --- |
| Remplissage, contenu de la baignoire = 50 Remplissage, contenu de la baignoire = 100 Remplissage, contenu de la baignoire = 150 Remplissage, contenu de la baignoire = 200 Remplissage, contenu de la baignoire = 250 Remplissage, contenu de la baignoire = 300 Remplissage, contenu de la baignoire = 350 Remplissage, contenu de la baignoire = 400 Remplissage, contenu de la baignoire = 450 Remplissage, contenu de la baignoire = 500 Remplissage, contenu de la baignoire = 550 Remplissage, contenu de la baignoire = 600 | Remplissage, contenu de la baignoire = 600 Fuite, contenu de la baignoire = 580 Remplissage, contenu de la baignoire = 630 Remplissage, contenu de la baignoire = 680 Remplissage, contenu de la baignoire = 730 Remplissage, contenu de la baignoire = 780 Remplissage, contenu de la baignoire = 830 Remplissage, contenu de la baignoire = 880 Remplissage, contenu de la baignoire = 930 Remplissage, contenu de la baignoire = 980 Remplissage, contenu de la baignoire = 1030 Baignoire enfin pleine ! |

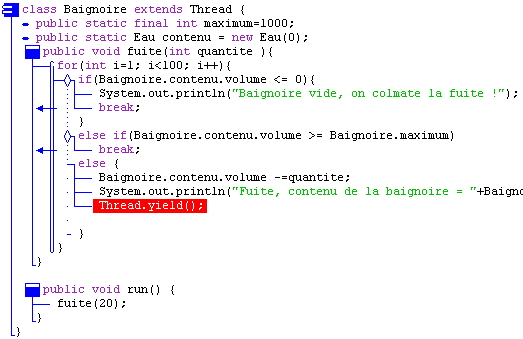
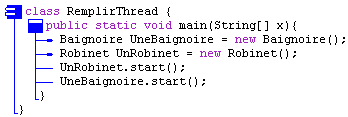
 ---- operation complete.  
  
Nous remarquons bien que le thread de remplissage  Robinet a été privilégié dans ses exécutions, puisque dans l'exécution précédente le thread Baignoire-fuite n'a pu exécuter qu'un seul tour de boucle.  
  
  
Alternons l'exécution de chaque thread  :

Nous souhaitons maintenant que le programme alterne le remplissage de la baignoire avec la fuite d'une façon équilibrée : action-fuite/action-remplissage/action-fuite/action-remplissage/...  
  
Nous utiliserons par exemple la méthode yield ( ) qui cesse temporairement l'exécution d'un thread et donc laisse un autre thread prendre la main. Nous allons invoquer cette méthode yield dans chacune des boucles de chacun des deux threads Robinet et Baignoire, de telle sorte que lorsque le robinet s'interrompt c'est la baignoire qui fuit, puis quand celle-ci s'interrompt c'est le robinet qui reprend etc... :

Voici le code de la classe Robinet :



Voici le code de la classe Baignoire :

  
  
Le reste du programme est inchangé :  
  
  
  
Résultats d'exécution obtenus :  
Remplissage, contenu de la baignoire = 50  
Fuite, contenu de la baignoire = 30  
Remplissage, contenu de la baignoire = 80  
Fuite, contenu de la baignoire = 60  
Remplissage, contenu de la baignoire = 110  
Fuite, contenu de la baignoire = 90  
............  
Remplissage, contenu de la baignoire = 980  
Fuite, contenu de la baignoire = 960  
Remplissage, contenu de la baignoire = 1010  
Baignoire enfin pleine !  
 ---- operation complete.  
  
Nous notons la parfaite alternance séquentielle entre les remplissage et les fuites. Ce petit exemple a permis de voir comment utiliser les threads pour simuler des déroulements d'actions parallèles.