Programmation Objet 2

2-nde partie

Didier FERMENT http://www.u-picardie.fr/ferment
Université de Picardie Jules Verne 2015-16

Après avoir étudié :

- La programmation d'interface graphique
- La programmation événementielle
- Le pattern Observeur-Observable
- L'architecture MVC
- Les java beans

Nous allons aborder :

- 4 techniques
  - Les exceptions
  - Les types génériques en Java
  - Le clonage
  - La sérialisation
- La programmation en environnement multi-thread :
  - thread, Thread et Runnable
  - Cycle de vie et interruption
  - Les variables partagées : synchonized, interblocage, les synchronisateurs, ...
  - Les collections en programmation multi-thread
  - Le pattern producteur-consommateur
  - L'exécution de taches : Callable, Future, Executor
  - Le multi-theading en programmation graphique

Les Exceptions (1/8)
Error et Exception

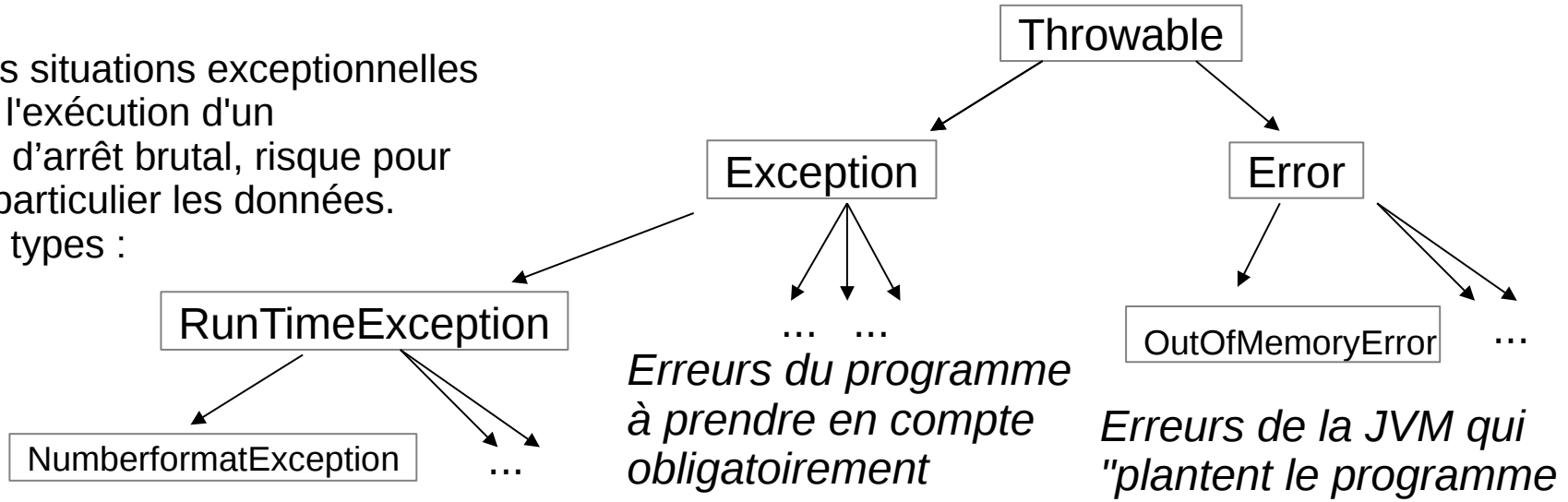
ErreurMemoire1.java

```
public class ErreurMemoire1 {
    public static void main(String[] args) {
      int beaucoup = Integer.parseInt(args[0]);
      Object[] tableau=new Object[beaucoup];
    }
}
```

```
$ java ErreurMemoire1 2000000000
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at ErreurMemoire1.main(ErreurMemoire1.java:4)
$ java ErreurMemoire1 abc
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
        at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:481)
        at java.lang.Integer.parseInt(Integer.java:514)
        at ErreurMemoire1.main(ErreurMemoire1.java:3)
```

Les erreurs sont des situations exceptionnelles qui mettent en péril l'exécution d'un programme : risque d'arrêt brutal, risque pour les ressources, en particulier les données. Java en distingue 3 types :

Non traitées, elles plantent l'exécution en cours.



Erreurs du programme

Erreurs du programme à prendre en compte obligatoirement

Erreurs de la JVM qui "plantent le programme

Les Exceptions (2/8)

Equation1.java

```
public class Equation1 {
  public static void main(String args[]) {
    int a = 0, b = 0;
    try {
      a = Integer.parseInt(args[0]);
      b = Integer.parseInt(args[1]);
    } catch (NumberFormatException nfe) {
      System.err.println("équation entière ax+b=0 : les paramètres a et b
                          doivent être entier");
      System.exit(0);
    }
    Integer x  = resoudreEquation(a,b);
    System.out.println("résultat équation entière ax+b=0 : X = "+x);
  }
  private static Integer resoudreEquation(int a, int b) {
    int sol = calculSolution(a,b);
    return new Integer(sol);
  }
  private static int calculSolution(int a, int b) {
    return b/a;
  }
}
```

RunTimeException

Les RunTimeExceptions comme ArithmeticException et NumberFormatException peuvent être traitées ou non. Exemple de traitement : try-catch

```
$ java Equation1 3 4
résultat équation entière ax+b=0 : X = 1
$ java Equation1 0 4
Exception in thread "main" java.lang.ArithmeticException: / by zero
       at Equation1.calculSolution(Equation1.java:19)
       at Equation1.resoudreEquation(Equation1.java:15)
       at Equation1.main(Equation1.java:11)
$ java Equation1 0 abc
équation entière ax+b=0 : les paramètres a et b doivent être entier
```

Les Exceptions (3/8)

Equation2.java

Try Catch

```
public class Equation2 {
  public static void main(String args[]) {
    ...
    try {
      Integer x  = resoudreEquation(a,b);
      System.out.println("résultat équation entière ax+b=0 : X = "+x);
    } catch (ArithmeticException ae) {
      System.err.println("équation entière ax+b=0 : erreur d\'éxécution : "
                         + ae.getMessage());
    }
  }
  private static Integer resoudreEquation(int a, int b) {
    int sol = calculSolution(a,b);
    return new Integer(sol);
  }
  private static int calculSolution(int a, int b) {
    return b/a;
  }
}
```

```
$ java Equation2 0 4
équation entière ax+b=0 : erreur d'éxécution : / by zero
```

Une Error ou une Exception est un Throwable : un "lançable" contenant un message/information sur le problème survenu. La méthode getMessage() fournit cette information.
L'exécution ne suit plus le cours normal des instructions de contrôle : if, for, ….
Le message lancé "va directement à la fin de la méthode actuelle" ou est capturé par un try-catch. Non capturé, il agit de même avec les méthodes appelantes jusqu'au .. main : l'exception est dite "propagée".

L'exception ArithmeticException est ici traitée/capturée via le blocs try/catch "essayer/attraper". Dès l'exception lancée, elle va exécuter le bloc catch lui correspondant.

Les Exceptions (4/8)

Equation3.java

Propagation / Capture

```
...
  Integer x  = resoudreEquation(a,b);
  if (x == null)
    System.out.println("résultat équation entière ax+b=0 : pas de solution");
  else
    System.out.println("résultat équation entière ax+b=0 : X = "+x);
}
private static Integer resoudreEquation(int a, int b) {
  try {
    int sol = calculSolution(a,b);
    return new Integer(sol);
  } catch (ArithmeticException ae) {
    return null;
  }
}
private static int calculSolution (int a, int b)   {
  return b/a;
}
}
```

capture

propagation

```
$ java Equation3 0 4
résultat équation entière ax+b=0 : pas de solution
```

La difficulté du programmeur :
Quand propage t'on une exception ?
A quel endroit capture t'on une exception ?
Que faire ? …. logger, afficher l'erreur, tenter une reprise, sauver ce qui est sauvegardable, ...

Les Exceptions (5/8)

Equation4.java
Equation4Bad.java

```
...
  private static Integer resoudreEquation(int a, int b) {
  try {
    int sol = calculSolution(a,b);
    return new Integer(sol);
  } catch (Exception e) {
    return null;
  }
}
private static int calculSolution (int a, int b) throws Exception  {
  // Exception trop large ! mais c'est pour l'exemple
  if (a == 0)
    throw new Exception("division par 0");
  else
    return b/a;
}
}
```

Lever une Exception

Changement de stratégie :
le programmer détecte une situation erronée
Il génère/lève/lance une exception par « throw »

Traiter les Exceptions "obligatoires"

Exception n'est pas une RunTimeException donc le compilateur oblige à traiter l'erreur :
soit la capture
soit la propagation assumée : méthode qui throws
donc propage l'exception

Sans l'instruction throws Exception :

```
$ javac Equation4Bad.java
Equation4Bad.java:28: unreported exception java.lang.Exception; must be caught or declared to be thrown
    throw new Exception("division par 0");
    ^
1 error
```

Les Exceptions (6/8)

Equation5.java

Plusieurs catch

```
public class Equation5 {
 public static void main(String args[]) {
   int a = 0, b = 0;
   try {
     a = Integer.parseInt(args[0]);
     b = Integer.parseInt(args[1]);
     Integer x  = resoudreEquation(a,b);
     System.out.println("résultat équation entière ax+b=0 : X = "+x);
   } catch (NumberFormatException nfe) {
     System.err.println("... les paramètres a et b doivent être entier"););
   } catch (Exception e) {
     System.out.println("résultat ...erreur ou pas de solution ");
   }
 }
 private static Integer resoudreEquation(int a, int b) throws Exception {
     int sol = calculSolution(a,b);
     return new Integer(sol);
 }
 private static int calculSolution (int a, int b) throws Exception  {
   // Exception trop large ! mais c'est pour l'exemple
   if (a == 0)
     throw new Exception("division par 0");
   else
     return b/a;
 }
}
```

Traitement par le premier catch correspondant au type d'exception.

Les Exceptions (7/8)

ExceptionFichier.java

finally

```
import java.io.*;
public class ExceptionFichier {
  public static void main(String args[]) {
    PrintWriter out = null;
    try {
      out = new PrintWriter(new FileWriter(args[0]));
      for (int i = 0; i < 100; i++)
        out.println("coucou : " + i);
    } catch (IOException ioe) {
      System.err.println("problème écriture fichier : " +  ioe.getMessage())
    } catch (Exception e) {
      System.err.println("erreur exécution: "+  e.getMessage());
    } finally {
      if (out != null)
        out.close();
      else
        System.out.println("fichier non ouvert");
    }
  }
}
```

```
$ java ExceptionFichier
erreur exécution: 0
fichier non ouvert
$ java ExceptionFichier coucou
```

La clause finally est exécutée dans tous les cas :
après un catch d'erreur
ou après le bloc try sans erreur.

Les IOExceptions sont à traiter obligatoirement.
Le schéma ci-dessus est à reprendre pour toute opération Open-Read/Write-Close sur des entrées-sorties.

Les Exceptions (8/8)

Pour aller plus loin ….

La méthode printStackTrace() de Throwable fournit l'ensemble des informations des méthodes appelées au moment de l'erreur.

La méthode getCause() permet d'obtenir l'erreur qui a provoquer la présente erreur dans le cas d'erreur en cascade, sinon null.

Le package java.util.logging fournit les outils pour logger les erreurs dans des fichiers.

Il est possible de créer ses propres exceptions en héritant d'Exception ou d'une de ses sous-classes, mais il y en a déjà certainement une qui vous convient.

Vers la généricité (0/13)
Autoboxing Unboxing

Autoboxing0.java

Ce n'est pas de la généricité mais cela aide le programmeur dans le même sens !

```
public class Autoboxing0 {
 ...
    List liste = new ArrayList();
    for (int i = 0; i < 10; i++)
      liste.add(new Integer(i));
    int le3eme = ((Integer)liste.get(2)).intValue();
    System.out.println("le3eme = "+le3eme);
    // et maintenant :
    for (int j = 11; j < 20; j++)
      liste.add(j);
    int le13eme = (Integer)liste.get(12);
    System.out.println("le13eme = "+le13eme);
```

```
$ java Autoboxing0
le3eme = 2
le13eme = 13
```

L'autoboxing permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant.
L'unboxing est l'opération inverse.

Reste un problème : la liste est d'objet pas uniquement d'Integer !
D'où la généricité !

Types Génériques (1/13)

Generic0.java

JAVA sans généricité

```
import java.util.*;
import java.io.*;
public class Generic0 {
  public static void main(String[] args) {
    ArrayList phrase = new ArrayList();
    phrase.add("il");
    phrase.add("fait");
    phrase.add("beau");
    System.out.println(phrase.toString());
    String le3emeMot = (String) phrase.get(2);
    System.out.println("le 3eme Mot : "+le3emeMot);
    phrase.add(new Integer(3));
    System.out.println(phrase.toString());
   }
}
```

```
$ java Generic0
[il, fait, beau]
le 3eme Mot : beau
[il, fait, beau, 3]
```

Il est possible de créer des classes de structures contenant n'importe quels type d'objet grâce :
aux collections (classes implémentant l'interface Collection)
et à la superclasse Object

Problèmes
- manipuler les objets référencés par ces structures oblige à faire du transtypage vers le bas (downcasting)
- risque d'erreur de cast, repérables uniquement à l'exécution (ClassCastException)
- impossibilité d'obliger à ce qu'une collection ne contienne qu'un seul type d'objet (comme avec les tableaux)

Types Génériques (2/13)

Generic1.java
Generic2.java

Utiliser les classes "génériques"

```
public class Generic1 {
  public static void main(String[] args) {
    ArrayList<String> phrase = new ArrayList<String>();
    phrase.add("il");
    phrase.add("fait");
    phrase.add("beau");
    System.out.println(phrase.toString());
    String le3emeMot = phrase.get(2);
    System.out.println("le 3eme Mot : "+le3emeMot);
   }
}
```

```
$ java Generic1
[il, fait, beau]
le 3eme Mot : beau
```

Utilisation de la classe "générique" : ArrayList<T>
appliquée au type String d'où ArrayList<String>
Inutile de faire de cast : ses méthodes renvoient le bon type.
Les vérifications se font à la compilation !

```
public class Generic2 {
  public static void main(String[] args) {
    ArrayList<String> phrase = new ArrayList<String>();
    phrase.add("il");
    ...
    phrase.add(new Integer(3));
    ...
```

```
$ javac Generic2.java
Generic2.java:12: error: no suitable method found for add(Integer)
….1 error
```

Types Génériques (3/13)

Generic3.java

Utiliser les classes "génériques"

```
ArrayList<String> phrase = new ArrayList<String>();
phrase.add("il");
phrase.add("fait");
phrase.add("beau");
System.out.println(phrase.toString());

String[] t = {};
String[] tab = phrase.toArray(t);

List<String> phrase2 = Arrays.asList(tab);
System.out.println(phrase2.toString());

List<String> phrase3 = Arrays.asList("il", "fait", "beau");
System.out.println(phrase3.toString());
```

```
$ java Generic3
[il, fait, beau]
[il, fait, beau]
[il, fait, beau]
```

Utilisation de la méthode toArray() de classe "générique" ArrayList<T>
public <T> T[ ] toArray(T[ ] tab) la méthode est générique : le paramètre tab ne sert qu'à fournir le type
Utilisation de la méthode static asList() de classe Arrays
public static <T> List<T> asList(T... a) la méthode est générique.

Remarque hors généricité : le paramètre … est un **vararg** : ça permet de passer un nombre non défini d'arguments d'un même type à une méthode. Le paramètre correspondant est un tableau de ce type.

Types Génériques (4/13)

Paire.java

Déclarer une classe "générique"

```
public class Paire <T> {
  private T premier, second;
  public Paire(T premier, T second) {
    this.premier = premier; this.second = second;
  }
  public String toString() { return "("+premier.toString()+","
                                   +second.toString()+")"; }
  public T getPremier() { return this.premier; }
  public T getSecond() { return this.second; }
  public void setPremier(T premier) { this.premier = premier; }
  public void setSecond(T second) {this.second = second; }
}
```

La notation <T> indique que la classe Paire utilise une variable de type :
T est le paramètre formel de type. Il peut y en avoir plusieurs.
La compilation de la classe "générique" donne simplement un fichier Paire.class

Convention de nommage pour les paramètres formels de type :
E - Element (de Collection)
K - Key
N - Number
T - Type
V - Value
S,U,V .... - 2, 3, 4éme types

```
public class Triplet <T,S,U> {
  private T premier, S second, U troisieme;
  public Triplet(T premier, S second, U troisieme)
  { ...
```

Types Génériques (5/13)

Paire.java
UsePaire1.java

Déclarer une classe "générique"

```
public class Paire <T> {
  private T premier, second;
  public Paire(T premier, T second) {
    this.premier = premier; this.second = second;
  }
 ...

public class UsePaire1 {
  public static void main(String[] args) {
    Paire<Double> point = new Paire<Double>(1.2, 4.5);
    point.setSecond(point.getPremier() + 3.4);
    System.out.println(point.toString());
...
```

```
$ java UsePaire1
(1.2,4.6)
```

Une classe générique doit être instanciée en passant/spécifiant le type.
Le type est n'importe quelle classe sauf :
les énumérations (car le type et les valeurs sont statiques)
les anonymous inner classes (car anonyme)
les exceptions (car c'est un mécanisme du runtime de la JVM)

Types Génériques (6/13)

UsePaire2.java
UsePaire3.java

Diamond, inférence de type, raw type

```
public class UsePaire2 {
  public static void main(String[] args) {
    Paire<Double> point1 = new Paire<Double>(1.2, 4.5);
    Paire<Double> point2 = new Paire<>(1.2, 4.5);
    Paire p3 = point2 ;
    System.out.println(p3.toString());
    // Paire<> p4 ;   compilation :  error: illegal start of type
...
```

Le "diamond" <> évite de répéter le type passé en utilisant l'inférence de type pour le déterminer.
Paire est le "raw type", le type de base, la classe de base d'une déclaration générique. Exemple : ArrayList est le raw type de ArrayList<T>

```
public class UsePaire3 {
  public static void main(String[] args) {
    List<Paire<Double>> listePoints = new ArrayList<>();
    listePoints.add(new Paire<Double>(1.2, 4.5));
    listePoints.add(new Paire<>(1.2, 4.5));
    System.out.println(listePoints.toString());
    Paire<Paire<Double>> pairePoints = new Paire<Paire<Double>>(
                          new Paire<Double>(1.2, 4.5), new Paire<>(1.2, 4.5) );
    System.out.println(pairePoints.toString());
   ...
```

Il est possible d'utiliser des génériques comme type passé.

```
$ java UsePaire3
[(1.2,4.5), (1.2,4.5)]
((1.2,4.5),(1.2,4.5))
```

Types Génériques (7/13)

Paire.java
UsePaire1.java
UsePaire4.java

L'Erasure

```
public class Paire <T> {
  private T Object premier, second;
  public Paire(T Object premier, T Object second) {
    this.premier = premier; this.second = second;
  }
  public T Object getPremier() { return this.premier; }
  public void setPremier(T Object premier) {
 ...
}
public class UsePaire1 {
  public static void main(String[] args) {
    Paire<Double> point = new Paire<Double>(1.2, 4.5);
    point.setSecond((Double)point.getPremier() + 3.4);
    ...
...
```

~~barré~~ = effacer
italique = ajouter

La compilation vérifie l'adéquation des types puis "efface" la partie générique pour ne laisser que le type raw et des casts.
Avantages :
- 1 seule classe compilée : le raw type
- pas de temps supplémentaire à l'exécution

Inconvénient :
- perte d''information sur le type passé

```
public class UsePaire4 {
  public static void main(String[] args) {
    Paire<Double> [] tabPoints = new Paire<Double>[2];
     // compilation : error: generic array creation
```

L'érasure efface le type, or un tableau doit contenir des informations sur le type de ses éléments.
Il n'est pas possible de créer des tableaux d'éléments génériques mais des collections oui.

Types Génériques (8/13)

MauvaisePaire.java

L'Erasure

```
public class MauvaisePaire <T> {
  private T premier, second;
  public MauvaisePaire(T premier, T second) {
    this.premier = premier; this.second = second;
  }
  public MauvaisePaire() {
    this.premier = this.second = new T();
  }
  public boolean contient(Object obj) {
    if (obj instanceof T) {
      T autre = (T)obj;
      return this.premier.equals(autre) ||
             this.second.equals(autre);
    } else
      return false;
  }
  public T[] asArray() {
    T[] tab = new T[2];
    tab[0] = this.premier; tab[1] = this.second;
    return tab;
  }
}
```

3 erreurs à la compilation conséquence de l'érasure :
- impossible d'instancier un objet sans son type
- impossible de vérifier un type qui n'existe pas
- impossible d'instancier tableau sans le type de ses éléments

Donc dans la classe générique :
ne pas instancier d'objet du type paramètre
ne pas utiliser l'opérateur instanceof avec le paramètre type
ne pas instancier de tableau dont les éléments sont du type paramètre

Types Génériques (9/13)

PaireOrdonnee.java
UsePaireOrdonnee.java

Type paramètre contraint

```
public class PaireOrdonnee <T extends Comparable> {
  private T premier, second;
  public PaireOrdonnee(T premier, T second) {
    if (premier.compareTo(second) < 0) {
      this.premier = premier;this.second = second;
    } else {
      this.premier = second;this.second = premier;
    }
  }
  ...
```

```
$ java UsePaireOrdonnee
PaireOrdonnee<Integer>(new Integer(4), new Integer(6))=(4,6)
PaireOrdonnee<String>("def", "abc")=(abc,def)
```

La notation <TypeParameter extends Class & Interface 1 & ... & Interface N >
introduit des contraintes sur le type paramètre :
- hériter d'1 classe
- implémenter des interfaces.

Cela réduit les types possibles
et permet d'utiliser les méthodes non static de la classe héritée et/ou des interfaces implémentées.
Les contraintes peuvent être de toutes classes, interfaces, et énumérations, voire de type paramètre (`PaireOrdonnee <T extends Comparable<T>>`)
sauf les types primitifs et les arrays.

Types Génériques (10/13)

Outils1.java
UseOutils1.java

Méthodes génériques

```
import java.util.*;
public class Outils1 {
  public static <T extends Comparable<T>> T max(T val1, T val2) {
    if (val1.compareTo(val2) > 0)
      return val1;
    else
      return val2;
  }
}
```

Les méthodes static et non static comme les constructeurs peuvent avoir des types paramètres.
Le ou les type paramètres doivent apparaître entre < > avant le type retour.
L'appel de telle méthode se fait comme les autres méthodes en utilisant l'inférence de type.

```
public class UseOutils1 {
  public static void main(String[] args) {
    Integer i = new Integer(1);
    Integer j = new Integer(2);
    System.out.println(Outils1.max(i,j));
   ...
```

```
$ java UseOutils1
2
```

Types Génériques (11/13)

Collections.java

Type paramètre wilcard

```
$ java Collections
[1, 2, 3]
[1, 2, 3]
```

```
public class Collections {
  public static <T> void copy ( List<? super T> dest,
                                List<? extends T> src) {
      for (int i=0; i<src.size(); i++)
        dest.add(src.get(i));
  }
  public static void main(String[] args) {
    ArrayList<Integer> nombres = new ArrayList<Integer>();
    nombres.add(1); nombres.add(2); nombres.add(3);
    System.out.println(nombres.toString());
    ArrayList<Number> copieNombres = new ArrayList<Number>();
    Collections.copy(copieNombres, nombres);
    System.out.println(copieNombres.toString());
  ...
```

La notation avec wilcard ? : <? super type > <? extends type > voire <?>
introduit une contrainte supérieure ou inférieure sur le type paramètre.
Comparaison avec les type paramètres contraints :
- n'introduit qu'une contrainte
- mais la contrainte peut être inférieure : <? super type >
- et la contrainte peut être relative à un type array.

HeritageGeneric1.java
HeritageGeneric2.java

Types Génériques (12/13)

Héritage et généricité

```
public class HeritageGeneric1 {
  public static void main(String[] args) {
    Paire<Double> point = new Paire<Double>(1.2, 4.5);
    Paire<Number> point2 = point;
   }
}
```

```
$ javac HeritageGeneric1.java
HeritageGeneric1.java:4: incompatible
types
found   : Paire<java.lang.Double>
required: Paire<java.lang.Number>
    Paire<Number> point2 = point;
 1 error                       ^
```

Piège :
Bien que Double hérite de Number, Paire<Double> n'hérite pas de Paire<Number>.
Mais ArrayList implémente List qui implémente Collection, donc
ArrayList<String> hérite de List<String> qui hérite Collection<String>.
Remarque : ArrayList<String> « hérite » de son raw type ArrayList

```
public class HeritageGeneric2 {
  public static void main(String[] args) {
    ArrayList<String> phrase = new ArrayList<String>();
    phrase.add("il"); phrase.add("fait");
    phrase.add("beau");
    List<String> phrase2 = phrase;
    Collection<String> phrase3 = phrase;
    System.out.println(phrase3.toString());
```

```
$ java HeritageGeneric2
[il, fait, beau]
```

Fin de la généricité (13/13)

Generic4.java

La boucle foreach

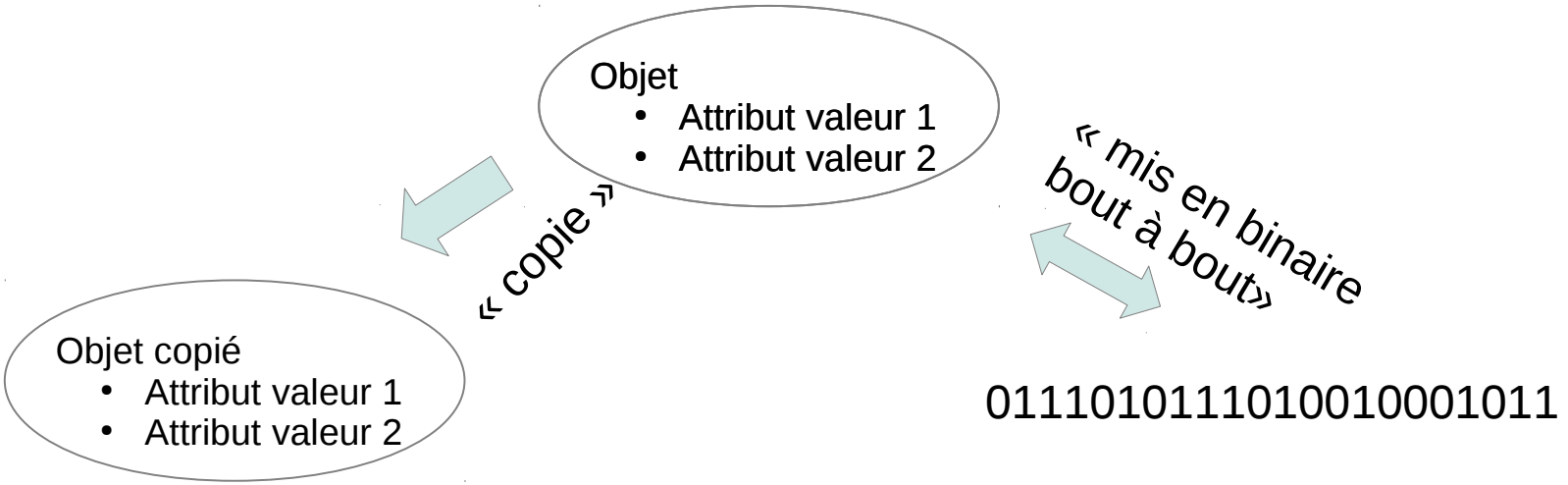
Ce n'est pas de la généricité mais cela aide le programmeur dans le même sens !

```
public class Generic4 {
  public static void main(String[] args) {
    ArrayList<String> phrase = new ArrayList<String>();
    phrase.add("il"); phrase.add("fait"); phrase.add("beau");
    String tout = "";
    for (Iterator<String> iter = phrase.iterator(); iter.hasNext(); )
    tout = tout + " " + iter.next();
    System.out.println("tout : "+tout);
    String tout2 = "";
    for (String mot : phrase)
      tout2 = tout2 + " " + mot;
    System.out.println("tout2 : "+tout2);
```

```
$ java Generic4
tout :  il fait beau
tout2 :  il fait beau
```

La classe Collection<T> implémente l'interface Iterable<T>
(la classe correspondante est Iterator)
qui permet d'utiliser la boucle "foreach".

Sérialisation Clonage (0/13)



Le clonage permet d'obtenir une copie d'un objet :
Utile pour conserver un état, pour la programmation parallèle, voire pour créer un objet complexe (car l'appel des constructeurs est coûteux en temps).

La sérialisation permet d'obtenir une copie sous forme binaire (voire en XML) d'un objet.
Utile pour conserver un objet dans un fichier ou le transmettre via le réseau. D'où la persistance et RMI.
De plus, l'objet peut être instancié à partir de cette copie sérialisée.

Ces 2 opérations pré-existent pour tout objet dans la JVM : les types primitifs doivent être clonés ou sérialisés de manière unique. Mais ces 2 opérations ne sont autorisées que si le programmeur le fait. Les 2 interfaces Serialization et Cloneable sont dites interface de marquage : elles servent à indiquer que l'opération est autorisée.

Les difficultés surviennent quand un objet est lui-même composé d'autres objets : comment appliquer "récursivement" le clonage ou la sérialisation ?

Sérialisation (1/13)
ObjectOutputStream

Puce1.java
Persistance0.java

```
public class Puce1 {
  String nom = null;
  int nombrePattes = -1;
  public Puce1(String n, int nb) {
    nom = n; nombrePattes = nb;
  }
  public String toString() {
    ...
```

```
$ java Persistance0
puce : Puce nom : pupuce, nombre de pattes = 6
java.io.NotSerializableException: Puce1
    at java.io.ObjectOutputStream.
       writeObject0(ObjectOutputStream.java:1180)
    at java.io.ObjectOutputStream.
       writeObject(ObjectOutputStream.java:346)
    at Persistance0.main(Persistance0.java:9)
```

```
public class Persistance0 {
  public static void main(String[] args)  {
    try {
      Puce1 puce =  new Puce1("pupuce", 6);
      System.out.println("puce : "+puce.toString());
      FileOutputStream fos = new FileOutputStream("sauvePuce");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(puce);
      oos.close();
    } catch (Exception e) {
      e.printStackTrace();
   ...
```

La classe ObjectOutputStream a une méthode writeObject() pour sérialiser un objet sérialisable : ce n'est pas le cas de Puce1 !

Sérialisation (2/13)
interface Serializable

```
import java.io.*;
public class Puce5 implements Serializable {
  String nom = null;
  int nombrePattes = -1;
  public Puce5(String n, int nb) {
    nom = n; nombrePattes = nb;
  }
  public String toString() {
    return "Puce nom : " + nom +", nombre de pattes = " + nombrePattes;
  }
}
```

Implémenter l'interface Serializable n'oblige à aucune implémentation de méthode !
C'est une interface de marquage : elle sert à indiquer que l'opération de sérialisation pré-existante est autorisée.

```
$ javac Puce5.java
$
```

La sérialisation, par défaut, encode chaque variable de type primitif et récursivement sérialise les références à des objets contenus. L'ensemble est mis bout à bout : en série.
Remarque : String implémente l'interface Serializable.

Ci-dessous la sérialisation de Puce5("pupuce", 4) :

```
$ od -c sauvePuce
0000000 254 355  \0 005   s   r  \0 005   P   u   c   e   5   n 274   (
0000020 312 241 332 337 330 002  \0 002   I  \0  \f   n   o   m   b   r
0000040   e   P   a   t   t   e   s   L  \0 003   n   o   m   t  \0 022
0000060   L   j   a   v   a   /   l   a   n   g   /   S   t   r   i   n
0000100   g   ;   x   p  \0  \0  \0 004   t  \0 006   p   u   p   u   c
0000120   e
0000121
```

Sérialisation (3/13)

Puce5.java
Persistance5.java

Persistance dans un fichier

```
public class Persistance5 {
  public static void main(String[] args)  {
    try {
      Puce5 puce =  new Puce5("pupuce", 6);
      System.out.println("puce : "+puce.toString());
      FileOutputStream fos = new FileOutputStream("sauvePuce");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(puce);
      oos.close();
      puce = null;
      System.out.println("puce : "+puce);
      FileInputStream fis = new FileInputStream("sauvePuce");
      ObjectInputStream ois = new ObjectInputStream(fis);
      puce = (Puce5) ois.readObject();
      ois.close();
      System.out.println("puce : "+puce.toString());

   ...
```

La classe ObjectOutputStream a une méthode writeObject() pour sérialiser un objet sérialisable dans un flot. Et la classe ObjectInputStream a une méthode readObject() pour de-sérialiser : il faut cast-er l'objet dans la bonne classe.

```
$ java Persistance5
puce : Puce nom : pupuce, nombre de pattes = 6
puce : null
puce : Puce nom : pupuce, nombre de pattes = 6
```

Sérialisation (4/13)
variable transient

Vect.java

```
import java.io.* ;
public class Vect implements Serializable {
  private Object[] tabElement = new Object[10];
  private transient int taille = 0;

  public Object elementAt(int index)
      throws ArrayIndexOutOfBoundsException {
    if (index >= taille)
      throw new ArrayIndexOutOfBoundsException(index);
    else
      return tabElement[index];
  }
  public void add(Object element) {
    if (tabElement.length == taille)
      resize(tabElement.length*2);
    tabElement[taille++] = element;
  }
  public String toString() {
    StringBuffer res = new StringBuffer("[ ");
    for (int i = 0 ; i < taille-1; ++i)
      res.append(tabElement[i]+", ");
    if (taille > 0)
      res.append(tabElement[taille-1]+" ");
    res.append("]");
    return res.toString();
  }
```

Une réécriture rapide et incomplète de Vector ….

Sérialisation (5/13)
… suite

Vect.java

```
  private void resize(int nouvelleTaille) {
    Object[] newTabElement = new Object[nouvelleTaille];
    System.arraycopy(tabElement, 0, newTabElement, 0, nouvelleTaille);
    tabElement = newTabElement;
    taille = nouvelleTaille;
  }
  private void writeObject(ObjectOutputStream flotOut)
          throws IOException {
    if (tabElement.length > taille)
      resize(taille);
    flotOut.defaultWriteObject();
  }
  private void readObject(ObjectInputStream flotIn)
        throws IOException, ClassNotFoundException
  {
    flotIn.defaultReadObject();
    taille = tabElement.length;
  }
}
```

Du fait de la réduction avant sérialisation, la taille correspond à celle du tableau.
Les variables transient ne sont pas sérialisées.

Pour redéfinir la sérialisation d'un objet, il faut implémenter des méthodes private writeObject() et readObject() bien qu'elles ne fassent parties d'aucune interface !

Les méthodes defaultWriteObject() et defaultReadObject() sont les méthodes de sérialisation par défaut.

Sérialisation (6/13)

TestVect1.java

```
public class TestVect1  {
  public static void main(String args[]) {
    try {
      Vect vect = new Vect();
      vect.add(new Puce5("pupuce", 6));
      vect.add(new Puce5("zezette", 5));
      ObjectOutputStream flotEcriture =
          new ObjectOutputStream(new FileOutputStream("Vect_Puce5"));
      flotEcriture.writeObject(vect);
      flotEcriture.close();
      ObjectInputStream flotLecture =
          new ObjectInputStream(new FileInputStream("Vect_Puce5"));
      Vect vect2 = (Vect)flotLecture.readObject();
      flotLecture.close();
      System.out.println("vect2 : "+vect2.toString());
    } catch (IOException ioe) {
      System.out.println(" erreur :" + ioe.toString());
    } catch (ClassNotFoundException cnfe) {
      System.out.println(" erreur :" + cnfe.toString());
    }
  }
}
```

La classe Vect est sérialisable !
Et récursivement Puce5
Et récursivement String

```
$ java TestVect1
vect2 : [ Puce nom : pupuce, nombre de pattes = 6,
Puce nom : zezette, nombre de pattes = 5 ]
```

Sérialisation (7/13)

TestVect2.java

Algorithme serialiser un objet :

- Pour chaque attribut :
  - Si transient
  - Alors rien
  - Sinon
    - Si type primitif
    - Alors opération prédéfinie
    - Sinon reférence d'autre objet
      - Si autre objet sérialisable
      - Alors récursiver sur autre objet
      - Sinon erreur

```
import java.io.*;
public class TestVect2  {
  public static void main(String args[]) {
    try {
      Vect vect = new Vect();
      vect.add(new Puce5("pupuce", 6));
      vect.add(new Puce1("zezette", 5));
      ObjectOutputStream flotEcriture =
          new ObjectOutputStream(new FileOutputStream("Vect_Puce1et5"));
      flotEcriture.writeObject(vect);
...
```

```
$ java TestVect2
 erreur :java.io.NotSerializableException: Puce1
```

Erreur normale : le mécanisme récursif de sérialisation tombe sur un objet non sérialisable : Puce1 .

Sérialisation (7bis/13)

Puce6.java
Persistance2.java

```
import java.beans.XMLDecoder;
import java.beans.XMLEncoder;
...
  public static void main(String args[]) {
    Puce6 puce = new Puce6();
    puce.setNom("pupuce");
    puce.setNombrePattes(6);
    try {
      System.out.println("puce : "+puce.toString());
      OutputStream out = new BufferedOutputStream(new FileOutputStream("sauve.xml"));
      XMLEncoder encoder = new XMLEncoder(out);
      encoder.writeObject(puce);
      encoder.close();
      puce = null;
      System.out.println("puce null");
      InputStream in = new BufferedInputStream(new FileInputStream("sauve.xml"));
      XMLDecoder decoder = new XMLDecoder(in);
      puce = (Puce6) decoder.readObject();
      decoder.close();
      System.out.println("puce : "+puce.toString());
    } catch (IOException ioe) {
...
```

La sérialisation XML ci-contre est ne concerne que les beans !

Il existe d'autres sérialisations XML comme par exemple : JAXB basé sur des POJOs et des annotations

C'est un bean !

```
public class Puce6 {
  private String nom = null;
  private int nombrePattes = 0;
  public Puce6() { }
  public String getNom() { return nom; }
  public void setNom(String n) {  this.nom = n; }
  public int getNombrePattes() {return nombrePattes; }
  public void setNombrePattes(int np ) {
                  if (np > 0) this.nombrePattes = np; }
  public String toString() {
    return "Puce nom : " + nom +", nombre de pattes = "
           + nombrePattes;
  }
}
```

```
$ java Persistance2
…
$ cat sauve.xml
 <?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_79"
      class="java.beans.XMLDecoder">
 <object class="Puce6">
  <void property="nom">
   <string>pupuce</string>
  </void>
  <void property="nombrePattes">
   <int>6</int>
  </void>
 </object>
</java>
```

Puce1.java
Clonage0.java

Clonage (8/13)
méthode clone()

```
public class Puce1 {
  String nom = null;
  int nombrePattes = -1;
    …
}
```

```
public class Clonage0 {
  public static void main(String[] args)  {
    Puce1 puceA =  new Puce1("pupuce", 6);
    Puce1 puceB ;
    puceB = puceA.clone() ;
  }
}
```

```
$ java Clonage0
Clonage0.java:6: error: clone() has protected access in
Object
    puceB = puceA.clone() ;
                  ^
Clonage0.java:6: error: incompatible types
    puceB = puceA.clone() ;
                  ^
  required: Puce1
  found:    Object
2 errors
```

La classe Object possède une méthode :
protected Object clone() throws CloneNotSupportedException

Le type retourné est Object

Pour l'utiliser, il faut la rendre public.

Puce2.java

Clonage (9/13)
interface Cloneable

```
public class Puce2 implements Cloneable {
  StringBuffer nom = null;
  int nombrePattes = -1;
  public Puce2(String n, int nb) {
    nom = new StringBuffer(n); nombrePattes = nb;
  }
  public String toString() {
    return "Puce nom : " + nom.toString()
           +", nombre de pattes = " + nombrePattes;
  }
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```

La classe Puce2 pour implémenter l'interface Cloneable doit redéfinir public la méthode clone().

Si l'objet n'implémente pas Cloneable, elle lance l'exception CloneNotSupportedException.

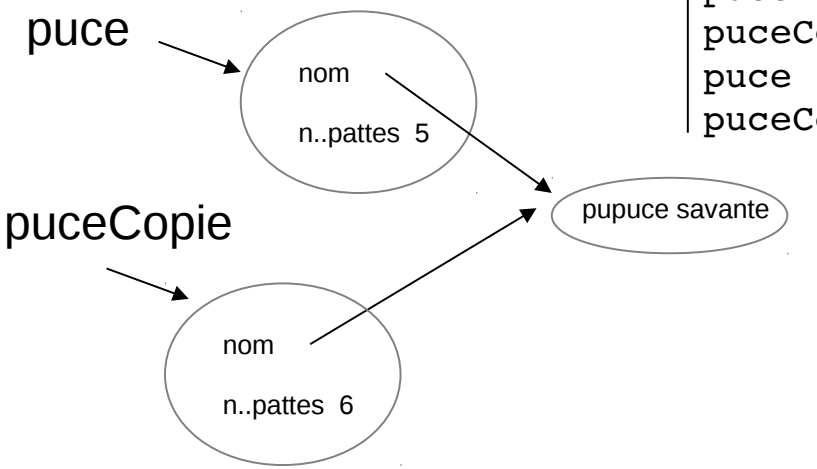
Elle adopte ici le comportement par défaut : elle copie les valeurs des variables donc les types primitifs et les références sur les Objets. Donc ce n'est pas, par défaut un algorithme récursif. La classe StringBuffer n'est pas Cloneable mais ça ne gène pas …

Puce2.java
Clonage2.java

Clonage (10/13)
clonage "en surface"

```
public class Clonage2 {
  public static void main(String[] args)  {
    try {
      Puce2 puce =  new Puce2("pupuce", 6);
      Puce2 puceCopie =  (Puce2) puce.clone();
      System.out.println("puce : "+puce.toString());
      System.out.println("puceCopie : "+puceCopie.toString());
      puce.nom.append(" savante");
      puce.nombrePattes = 5;
      System.out.println("puce : "+puce.toString());
      System.out.println("puceCopie : "+puceCopie.toString());
    } catch (CloneNotSupportedException cnse) {
      System.out.println("puce non clonable");
    }
}
```

```
$ java Clonage2
puce : Puce nom : pupuce, nombre de pattes = 6
puceCopie : Puce nom : pupuce, nombre de pattes = 6
puce : Puce nom : pupuce savante, nombre de pattes = 5
puceCopie : Puce nom : pupuce savante, nombre de pattes = 6
```



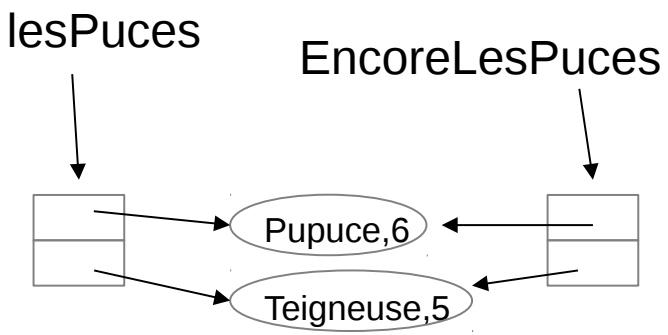
La classe Puce2 utilise en fait la méthode super.clone(), donc le comportement par défaut :
en particulier, la copie simple des références d'objets.
La variable d'instance nom de l'objet et son clone sont donc la même référence.

Puce1.java
Clonage1.java

Clonage (11/13)
le clonage des collections JAVA

```
public class Clonage1 {
  public static void main(String[] args)  {
    Puce1 puceA =  new Puce1("pupuce", 6);
    Puce1 puceB =  new Puce1("teigneuse", 5);
    ArrayList<Puce1> lesPuces = new ArrayList<Puce1>();
    lesPuces.add(puceA); lesPuces.add(puceB);
    ArrayList<Puce1> encoreLesPuces = (ArrayList<Puce1>) lesPuces.clone();
    lesPuces.remove(puceB);
    lesPuces.get(0).nom = "zezette";
    System.out.println("lesPuces : "+lesPuces.toString());
    System.out.println("encoreLesPuces : "+encoreLesPuces.toString());
   ...
```

```
$ java Clonage1
lesPuces : [Puce nom : zezette, nombre de pattes = 6]
encoreLesPuces : [Puce nom : zezette, nombre de pattes = 6,
Puce nom : teigneuse, nombre de pattes = 5]
```



La classe ArrayList est Cloneable donc possède une méthode clone() qui opère ainsi : elle duplique la liste de références sur les objets, mais elle ne duplique pas les objets de la liste.

Rmq : les arrays son Cloneable selon la même stratégie.

Puce3.java
Clonage3.java

Clonage (12/13)
clonage "en profondeur"

```
public class Puce3 implements Cloneable {
...
  public Object clone() throws CloneNotSupportedException {
    Puce3 copie = (Puce3) super.clone();
    copie.nom = new StringBuffer(this.nom.toString());
    return copie;
  }
...
```

```
$ java Clonage3
puce : Puce nom : pupuce, nombre de pattes = 6
puceCopie : Puce nom : pupuce, nombre de pattes = 6
puce : Puce nom : pupuce savante, nombre de pattes = 5
puceCopie : Puce nom : pupuce, nombre de pattes = 6
```

La classe Puce3 clone en profondeur :
l'objet référencé dans sa variable d'instance nom est dupliqué.
Le clonage en profondeur consiste à appliquer récursivement le clonage à l'intérieur de l'objet … jusqu'à un certain niveau.

Rappel : la classe StringBuffer n'est pas Cloneable.

Clonage (13/13)
clonage "en profondeur"

```
public class Clonage4 {
  public static void main(String[] args)  {
    try {
      Puce3 puceA =  new Puce3("pupuce", 6);
      Puce3 puceB =  new Puce3("teigneuse", 5);
      SuperList lesPuces = new SuperList();
      lesPuces.add(puceA); lesPuces.add(puceB);
      SuperList encoreLesPuces = (SuperList) lesPuces.clone();
      lesPuces.remove(puceB);
      puceA.nom.reverse();
      System.out.println("lesPuces : "+lesPuces.toString());
      System.out.println("encoreLesPuces : "+encoreLesPuces.toString());
    } catch (CloneNotSupportedException cnse) {
      System.out.println(cnse.getMessage());
    }

  }
}
class SuperList extends ArrayList<Puce3> {
  public Object clone() throws CloneNotSupportedException {
    SuperList copie = new SuperList();
    for (Puce3 puce : this)
      copie.add((Puce3)puce.clone());
    return copie;
  }
}
```

La classe ArrayList clone "en surface". Elle est étendue en redéfinissant la méthode clone() en "profondeur".

```
$ java Clonage4
lesPuces : [Puce nom : ecupup, nombre de pattes = 6]
encoreLesPuces : [Puce nom : pupuce, nombre de pattes = 6,
Puce nom : teigneuse, nombre de pattes = 5]
```

Thread (1/38)

Moyenne.java
TestLent.java

Paralléliser

```
public class Moyenne ...
    Scanner scanIn = new Scanner(System.in);
    double total = 0.0; int nombre = 0;
    while (scanIn.hasNextDouble()) {
      total += scanIn.nextDouble();
      nombre ++;
    }
    System.out.println("Moyenne des "+nombre
    +" nombres = "+(total/nombre));
...
```

```
$ java Moyenne
4,6
8,9
=
Moyenne des 2 nombres = 6.75
```

Problème : chacun passe beaucoup de temps bloqué en attente de lecture
Comment faire ces 2 taches "en même temps" ?

Solution avec 2 processus : 1 fenêtre "Shell" pour chacun
Autre solution puisqu'ils sont écrit dans le même langage : 2 threads du même processus car le changement de contexte de thread est moins long que celui de processus

```
class TestLent ….
  InputStream ins = null;
  try {
    URL url = new URL("http://www.u-picardie.fr/ferment/chargement_lent.php");
    ins = url.openStream();
    BufferedReader buffReader = new BufferedReader(new InputStreamReader(ins));
    String ligne;
    while ((ligne = buffReader.readLine()) !=  null)
      System.out.println(ligne);
  }
  catch (MalformedURLException e) {}
  catch (IOException e) {}
  finally {
    if (ins != null)
      try {
        ins.close();
      } catch (IOException e) {}
  }
```

```
$ java TestLent
chargement tres lent
chargement tres tres lent
chargement tres tres tres lent
chargement tres tres tres tres lent
chargement tres tres tres tres tres lent
chargement tres tres tres tres tres tres lent
...
```

Thread (2/38)
Runnable

Lent.java

```
public class Lent implements Runnable {
  public void run() {
    InputStream ins = null;
    try {
      URL url = new URL("http://www.u-picardie.fr/ferment/chargement_lent.php");
      ins = url.openStream();
      BufferedReader buffReader = new BufferedReader(new InputStreamReader(ins));
      String ligne;
      while ((ligne = buffReader.readLine()) !=  null)
        System.out.println(ligne);
    }
    catch (MalformedURLException e) {}
    catch (IOException e) {}
    finally {
      if (ins != null)
        try {
          ins.close();
        } catch (IOException e) {}
    }
  }
}
```

Un thread, un fil d'exécution, peut être vu comme un "processus léger".
En Java, les objets à exécuter sur un fil d'exécution propre doivent implémenter l'interface Runnable : la méthode run() sera exécutée comme thread par la JVM avec sa propre pile d'exécution.

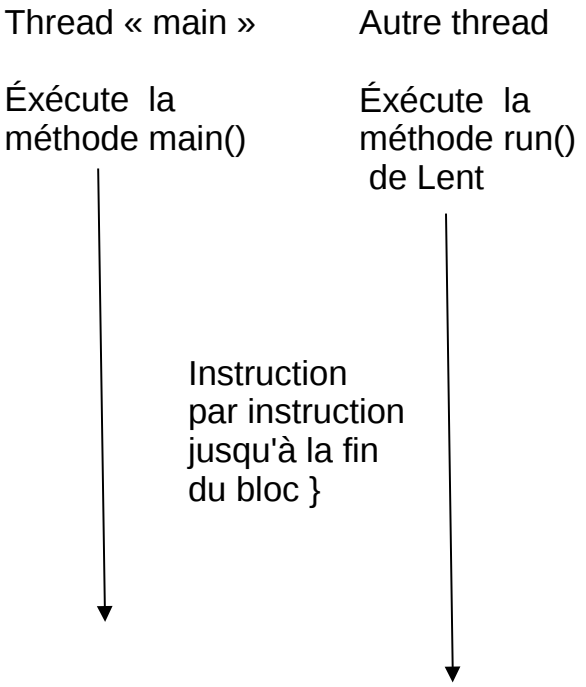
Remarque : toute application Java fonctionne avec au moins 1 processus qui exécute au moins 1 thread chargé de la méthode main().

Thread (3/38)

DeuxTachesParalleles.java
Lent.java

Thread

```
public class DeuxTachesParalleles {
  public static void main (String [] args) {
    Scanner scanIn = new Scanner(System.in);
    double total = 0.0; int nombre = 0;
    Thread autreTache = new Thread(new Lent());
    autreTache.start();
    while (scanIn.hasNextDouble()) {
      total += scanIn.nextDouble();
      nombre ++;
    }
    System.out.println("Moyenne des "+nombre+" nombres = "+(total/nombre));
  }
}
```



La classe Thread comporte plusieurs méthodes d'administration des threads (fil d'exécution d'un Runnable).
En particulier, la méthode start() qui demande à la JVM un nouveau fil d'exécution pour la méthode run .
Les 2 threads (fil d'exécution) sont bien indépendants car l'un peut s'arrêter avant l'autre.

```
$ java DeuxTachesParalleles
chargement tres lent
chargement tres tres lent
chargement tres tres tres lent
4,chargement tres tres tres tres lent
5chargement tres tres tres tres tres lent
chargement tres tres tres tres tres tres lent
6,chargement tres tres tres tres tres tres tres lent
7chargement tres tres tres tres tres tres tres tres lent
chargement tres tres tres tres tres tres tres tres tres lent
=chargement tres tres tres tres tres tres tres tres tres tres lent
Moyenne des 2 nombres = 5.6
```

Thread (4/38)
la classe Thread

DeuxTachesParalleles2.java

```
public class DeuxTachesParalleles2 {
  ...
    Thread autreTache = new Thread(new Lent());
    autreTache.start();
    System.out.println("autreTache: "+autreTache.toString());
    System.out.println("Thread.currentThread(): "+Thread.currentThread().toString());
    while (scanIn.hasNextDouble()) {
      total += scanIn.nextDouble();
      nombre ++;
    ...
```

```
$ java DeuxTachesParalleles2
autreTache: Thread[Thread-0,5,main]
Thread.currentThread(): Thread[main,5,main]
chargement tres lent
...
```

La méthode toString() de la classe Thread affiche le nom du thread, son niveau de priorité et le nom de son groupe.
La méthode statique currentThread() fournit le présent thread en train d'exécuter la méthode.

Remarque : la classe Thread implémente l'interface Runnable et donne un moyen de (re)définir la méthode run.
Risque de confusion entre Thread (classe, objet de) et le thread fil d'exécution !

Thread (5/38)
Vie d'un thread

TroisTachesParalleles.java
Boucleinfinie.java
Dodo.java

```
public class BoucleInfinie implements Runnable {
  public void run() {
    boolean encore = true;
    int j = 1;
    while (encore) {
      for (int i=0; i<2000000000; ++i)
        j = j*i ;
      System.out.println("....2000000000");
    }
  }
}
```

```
public class Dodo implements Runnable {
  public void run()  {
    try {
      for (int i=0; i<20; i++) {
        System.out.println("dodo");
        Thread.sleep(500);
      }
     } catch (InterruptedException ie) { }
  }
}
```

```
public class TroisTachesParalleles {
  public static void main (String [] args) {
    boolean encore = true;
    Scanner scanIn = new Scanner(System.in);
    Thread tacheDodo = new Thread(new Dodo());
    Thread tacheInfinie = new Thread(new BoucleInfinie());
    tacheDodo.start();
    tacheInfinie.start();
    while (encore) {
      String frappe = scanIn.next();
      if (frappe.equals("q"))
         encore = false;
    }
...
```

La vie de chaque thread s'organise autour de 3 états principaux :
- en cours d'exécution
- ne peut pas s'exécuter (ex: sleep)
- exécutable

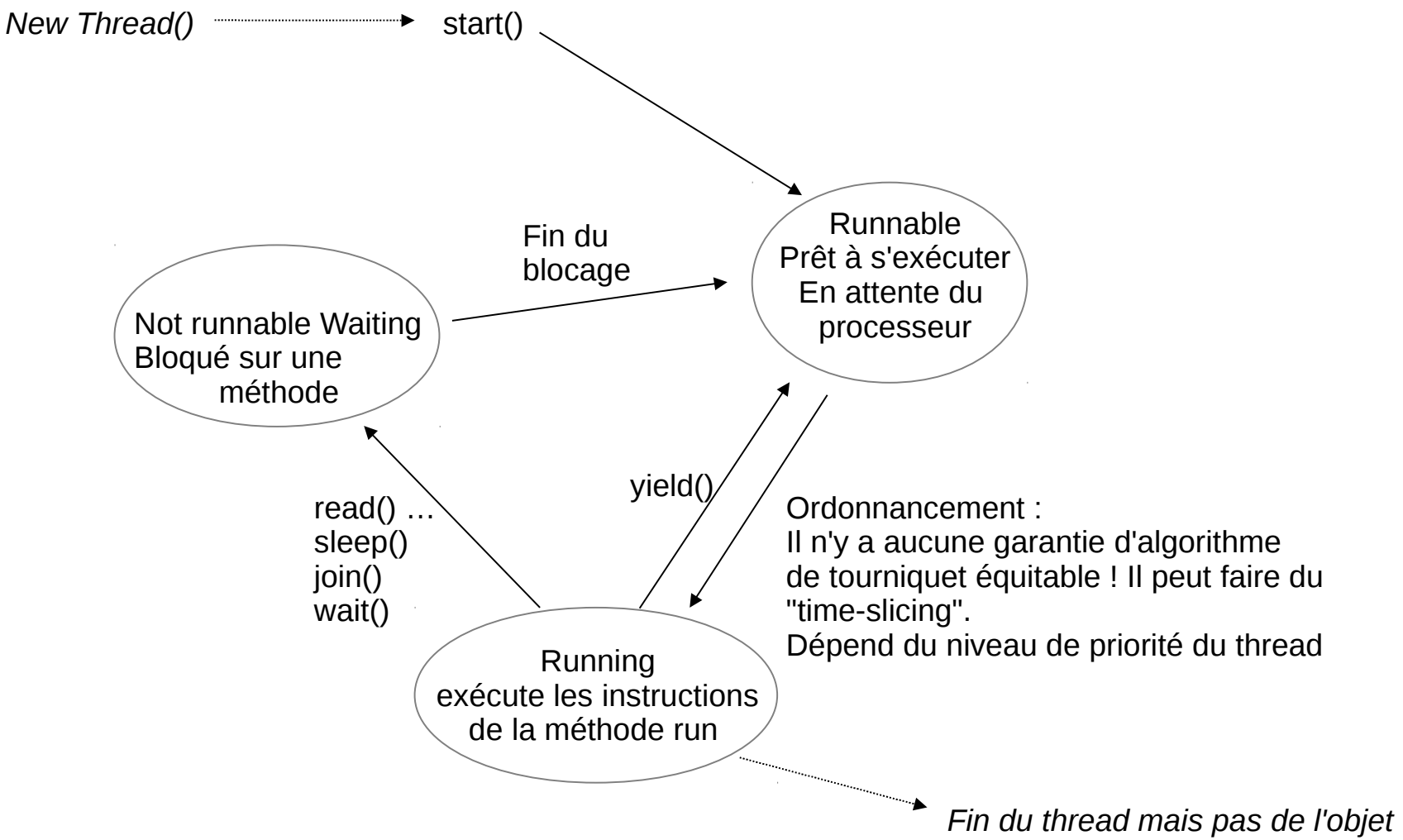
Un ordonnanceur de la JVM "arbitre" entre les différents threads.

La méthode sleep() suspend un thread pendant des millisecondes. Cette méthode bloquante peut être interrompue par l'exception InterruptedException.

La frappe de "q" pour quitter n'arrête que le thread main. Il faut "killer" le processus pour arrêter les 2 autres.

Thread (6/38)

Cycle de vie d'un thread



Thread (7/38) : Comment arrêter un thread ? interrupt()

Arret1.java
Boucleinfinie.java
Dodo.java

```
import java.util.*;
public class Arret1 {
  public static void main (String [] args) {
    boolean encore = true;
    Scanner scanIn = new Scanner(System.in);
    Thread tacheDodo = new Thread(new Dodo());
    Thread tacheInfinie = new Thread(new BoucleInfinie());
    tacheDodo.start();
    tacheInfinie.start();
    while (encore) {
      String frappe = scanIn.next();
      if (frappe.equals("q")) {
         encore = false;
         tacheDodo.interrupt();
         tacheInfinie.interrupt();
      }
    }
  }
}
```

```
$ java Arret1
dodo
dodo
dodo
dodo
dodo
dodo
dodo
dodo
....2000000000
dodo
dodo
dodo
dodo
dodo
qdodo

....2000000000
....2000000000
....2000000000
....2000000000
```

L'arrêt d'un thread peut se faire :
normalement en fin de méthode run
"brusquement" par une RuntimeException ou une Error
ou l'arrêt du processus.

La méthode interrupt() "demande" au thread de s'arrêter ...
Ça fonctionne avec le thread "Dodo" mais pas avec le thread "boucleInfinie" .

Boucleinfinie2.java
Dodo2.java
Arret2.java

Thread (8/38) : Comment arrêter un thread ? isInterrupted()

```
public class BoucleInfinie2 implements Runnable {
  public void run() {
    boolean encore = true;
    int j = 1;
    while (encore && ! Thread.currentThread().isInterrupted()) {
      for (int i=0; i<2000000000; ++i)
        j = j*i ;
      System.out.println("....2000000000");
    }
  }
}
```

La méthode isInterrupted() renseigne sur l'état interrompu ou non du thread.

Attention à la méthode interrupted() :
elle renvoie aussi l'état "interrompu ou non du thread" et elle inverse l'état.
Elle sert souvent à effacer une demande d'interruption.

```
public class Dodo2 implements Runnable {
  public void run() {
    try {
      for (int i=0; i<20 && ! Thread.currentThread().isInterrupted() ; i++) {
        System.out.println("dodo");
        Thread.sleep(500);
      }
    } catch (InterruptedException ie) { /* fin */}
  }
}
```

Thread (9/38) : Comment arrêter un thread ? InterruptedException

Dodo2.java
Arret2.java

```
public class Dodo2 implements Runnable {
  public void run() {
    try {
      for (int i=0; i<20 && ! Thread.currentThread().isInterrupted() ; i++) {
        System.out.println("dodo");
        Thread.sleep(500);
      }
    } catch (InterruptedException ie) { /* fin */}
  }
}
```

Lorsque une méthode interrupt() "demande" au thread de s'arrêter …

Soit le thread est sur une méthode bloquante interruptible par l'exception InterruptedException (exemple : sleep, wait, …) ,
alors l'exception est lancée donc try-catchée et l'état "isInterrupted" passe à true,

Soit le thread est sur une méthode bloquante non interruptible par l'exception InterruptedException (exemple : read sur socket, lock de verrou, select de Selector …),
ou le thread est sur des instructions et méthodes non bloquantes,
alors l'état "isInterrupted" passe à true.

Dans le thread de "Dodo", si interrupt() se produit pendant la méthode bloquante sleep alors l'interruption est lancée puis catch-ée à la fin de la méthode run() donc fin.
Par contre, très rarement, si interrupt() se produit ailleurs, l'état "isInterrupted" passe à true, donc la boucle stoppe.

VariablePartagee1.java

Variables partagées

```
public class VariablePartagee1 {
  private int compteur = 0;
  public static void main (String [] args) throws InterruptedException {
    VariablePartagee1 vp = new VariablePartagee1();
    vp.go();
  }
  public void go() throws InterruptedException {
    Thread increment1 = new Thread(new Incrementation());
    Thread increment2 = new Thread(new Incrementation());
    increment1.start();
    increment1.join();
    increment2.start();
    increment2.join();
    System.out.println("Fin : compteur =  "+compteur);
    compteur = 0;
    increment1 = new Thread(new Incrementation());
    increment2 = new Thread(new Incrementation());
    increment1.start();
    increment2.start();
    increment1.join();
    increment2.join();
    System.out.println("Fin : compteur =  "+compteur);
  }
  class Incrementation implements Runnable {
    public void run() {
      for (int i=0; i<10000000; ++i)
        compteur++;
    }
  }
}
```

La méthode bloquante join() attends la fin d'exécution du thread indiqué. Elle peut être interrompue par l'InterruptedException.

Dans le 1er cas (exécution séquentielle des 2 threads), la variable partagée compteur est finalement incrémentée de la bonne somme.

Ce n'est pas le cas dans la 2nde : exécution parallèle des 2 threads)

```
$ java VariablePartagee1
Fin : compteur =  20000000
Fin : compteur =  15664865
```

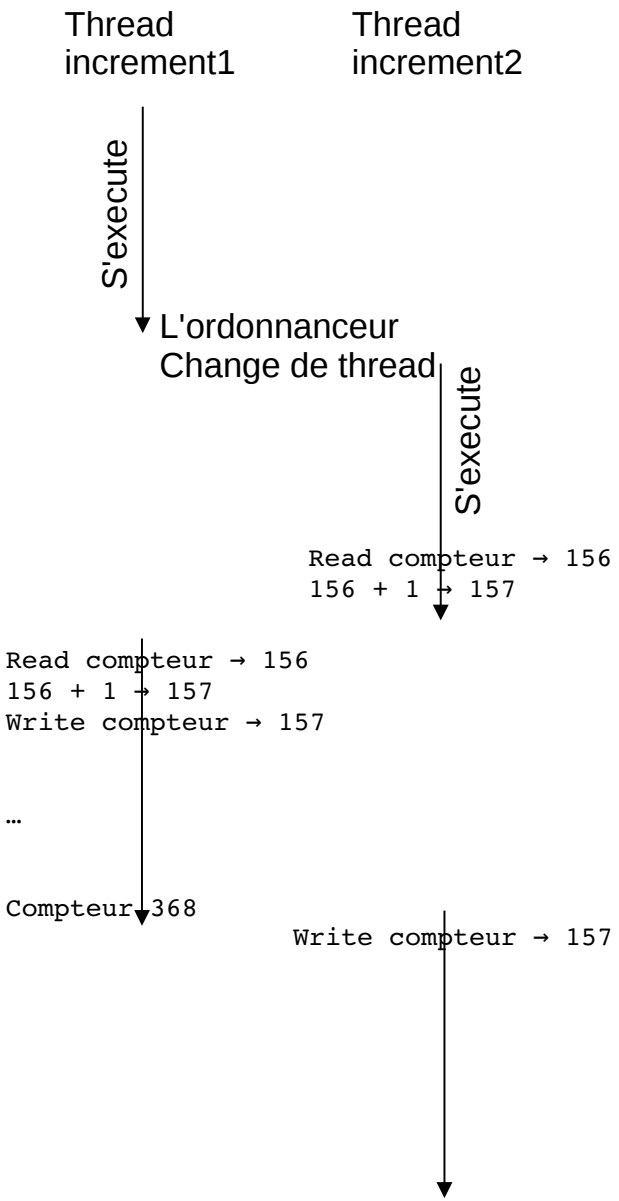
Thread (10bis/38)

VariablePartagee1.java

Variables partagées

```
…
    compteur = 0;
    increment1 = new Thread(new Incrementation());
    increment2 = new Thread(new Incrementation());
    increment1.start();
    increment2.start();
    increment1.join();
    increment2.join();
    System.out.println("Fin: compteur="+compteur);
  }
  class Incrementation implements Runnable {
    public void run() {
      for (int i=0; i<10000000; ++i)
        compteur++;
    }
  }
}
```

L'exécution parallèle des 2 threads) car l'instruction compteur++ est composée d'au moins 3 opérations dans le bytecode Java. Si l'ordonnanceur change de thread au milieu de ces opérations …..



Thread (11/38) : Variables partagées synchronized

VariablePartagee2.java

```
public class VariablePartagee2 {
  class Compteur {
    public int val = 0;
  }
  ...
  public void go() throws InterruptedException {
    Thread increment1 = new Thread(new Incrementation());
    Thread increment2 = new Thread(new Incrementation());
    increment1.start();
    increment1.join();
    increment2.start();
    increment2.join();
    System.out.println("Fin : compteur =  "+compteur.val);
    compteur = new Compteur();
    increment1 = new Thread(new Incrementation());
    increment2 = new Thread(new Incrementation());
    increment1.start();
    increment2.start();
    increment1.join();
    increment2.join();
    System.out.println("Fin : compteur =  "+compteur.val);
  }
  class Incrementation implements Runnable {
    public void run() {
      for (int i=0; i<10000000; ++i)
      synchronized(compteur) {
        compteur.val++;
      }
    }
  }
```

```
$ java VariablePartagee2
Fin : compteur =  20000000
Fin : compteur =  20000000
```

L'utilisation de l'instruction de contrôle synchronized résout le problème mais l'exécution prend plus de temps.

Chaque objet possède un verrou :
Pour "entrer" dans le bloc { compteur.val++; } le thread doit acquérir le verrou de compteur;
attente bloquante ;
tant que qu'il exécute le bloc, il possède et est le seul;
A la fin du bloc, il déverrouille;
Un autre thread en attente peut alors acquérir le verrou.

L'exclusion d'accès à l'incrémentation de la variable partagée est assurée.

Thread (12/38) : Variables partagées méthode synchronized

VariablePartagee3.java

```
public class VariablePartagee3 {
  class Compteur {
    public int val = 0;
    public synchronized void plusplus() {
      compteur.val++;
    }
  }
  ...
  class Incrementation implements Runnable {
    public void run() {
      for (int i=0; i<10000000; ++i)
        compteur.plusplus();
    }
  }
}
```

```
$ java VariablePartagee3
Fin : compteur =  20000000
Fin : compteur =  20000000
```

Une méthode synchronized est équivalente à synchronized(this).

Le mécanisme de verrou interne est ré-entrant :
un thread ayant un verrou peut entrer dans un autre bloc synchronized sur le même verrou

```
synchronized(obj) {
   ...
   synchronized(obj) {
     …
   }
}
```

L'attente bloquante sur synchronized n'est pas interrompue par l'InterruptedException.
L'attente du verrou interne n'est pas pas garantie contre le risque de "famine" : la spécification de la JVM n'impose rien sur le choix du thread à satisfaire lors de la libération du verrou.

Thread (13/38) : Interblocage par synchronized

Interblocage.java

```
public class Interblocage {
  ...
  private CompteBancaire compteA = new CompteBancaire("blingbling", 2000000000.0);
  private CompteBancaire compteB = new CompteBancaire("travailleurPlus", 1000.0);
  ...
    Thread redistrib1 = new Thread(new Redistribution(compteA, compteB));
    Thread redistrib2 = new Thread(new Redistribution(compteB, compteA));
    redistrib1.start();
    redistrib2.start();
    redistrib1.join();
    redistrib2.join();
    System.out.println("Fin travaillerPlusPourGagnerMoins");
  }
  class Redistribution implements Runnable {
    private CompteBancaire compte1;
    private CompteBancaire compte2;
    public Redistribution(CompteBancaire c1, CompteBancaire c2) {
      compte1 = c1; compte2 = c2;
    }
    public void run() {
      for (int i=0; i<200; ++i)
        synchronized(compte1) {
          if (compte1.solde > 10.0)
            synchronized(compte2) {
              compte2.solde += 10.0;
              compte1.solde -= 10.0;
            }
        }
    }
  }
```

Pour faire des opérations sur les comptes, il faut bien acquérir les verrous des comptes !

```
$ java Interblocage
Fin travaillerPlusPour...
$ java Interblocage
Fin travaillerPlusPour...
$ java Interblocage
Fin travaillerPlusPour...
$ java Interblocage
<ctrl>C
```

Thread (14/38)

trace de l'interblocage

```
$ java Interblocage
Fin ravaillerPlusPourGagnerMoins
$ java Interblocage
^\2012-07-05 12:20:56

Full thread dump OpenJDK Server VM (20.0-b11 mixed mode):

"Thread-1" prio=10 tid=0x8c129400 nid=0x163b waiting for monitor entry [0x8ba15000]
   java.lang.Thread.State: BLOCKED (on object monitor)
     at Interblocage$Redistribution.run(Interblocage.java:35)
     - waiting to lock <0xa9b84900> (a Interblocage$CompteBancaire)
     - locked <0xa9b84918> (a Interblocage$CompteBancaire)
     at java.lang.Thread.run(Thread.java:679)

"Thread-0" prio=10 tid=0x8c127800 nid=0x163a waiting for monitor entry [0x8ba66000]
   java.lang.Thread.State: BLOCKED (on object monitor)
     at Interblocage$Redistribution.run(Interblocage.java:35)
     - waiting to lock <0xa9b84918> (a Interblocage$CompteBancaire)
     - locked <0xa9b84900> (a Interblocage$CompteBancaire)
     at java.lang.Thread.run(Thread.java:679)

     ....


Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x08839e90 (object 0xa9b84900, a Interblocage$CompteBancaire),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x08837050 (object 0xa9b84918, a Interblocage$CompteBancaire),
  which is held by "Thread-1"
...
```

Envoyer un SIGQUIT à la JVM pour l'arrêter avec trace :
<ctrl>\ ou kill -3 en Unix
<ctrl><break>

Eviter l'interblocage est la tache la plus difficile pour programmer "thread-safe" …
Un moyen possible est fournit par la classe Lock avec sa méthode tryLock() qui bloque selon un paramètre délai.

Thread (15/38)

Autres risques de partage

Problème :
L'accès aux variables de type long et double n'est pas atomique : il nécessite 2 instructions de byte-code.
Solution :
le package java.util.concurrent.atomic qui fournit une classe AtomicLong

Problème :
L'optimisation du code conduit à stocker des valeurs de variables en cache avant affectation en mémoire. Donc la valeur en mémoire n'est pas forcément la bonne !
Solution :
Déclarer la variable volatile pour empêcher la mise en cache de sa valeur.

Problème :
Le mécanisme de verrou interne est basé sur le volontariat des programmeurs. Un autre thread pourrait accéder faire compteur.val sans, au préalable, acquérir le verrou compteur.
Solution :
Utiliser le pattern moniteur de Java

```
public class ClasseMonitorée {
  private final Object monVerrou = new Object();
  @GuardedBy("monVerrou") type objetGardé;
  void méthode() {
    synchronized(monVerrou) {
      // accès à objetGardé
    }
  }
}
```

L'annotation GuardedBy est pratique pour hériter !

Problème :
L'optimisation du code peut changer l'ordre d'exécution des instructions qui semblent indépendantes.
Solution :
Connaître les règles du MMJ modèle de mémoire de la JVM !

Thread (16/38) : partage thread-safe de variable : AtomicNumber

NombresAtomiques.java
CalculComplique3.java

```
import java.util.concurrent.atomic.AtomicInteger;
public class CalculComplique3 implements Runnable {
  private int x;
  private AtomicInteger result;
  public CalculComplique3(int x, AtomicInteger r) {
    this.x=x; this.result=r;
  }
  public void run() {
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
     result.addAndGet(y);
  }
}

public class NombresAtomiques {
  ...
  Scanner scanIn = new Scanner(System.in);
  int nombre1 = scanIn.nextInt();
  int nombre2 = scanIn.nextInt();
  AtomicInteger resultat = new AtomicInteger(0);
  Thread tache1 = new Thread(new CalculComplique3(nombre1, resultat));
  Thread tache2 = new Thread(new CalculComplique3(nombre2, resultat));
  tache1.start(); tache2.start();
  tache1.join(); tache2.join();
  System.out.println("Somme de CalculComplique3 de "+nombre1
                     +" et "+nombre2+" = "+resultat.intValue());
 . . .
```

Les variables "atomiques" du package java.util.concurrent.atomic répondent aux manques d'opérations atomiques sur les variables volatiles.

Ses méthodes addAndGet(), incrementAndGet(), compareAndSet() sont toutes effectuées en bloc synchonized(this).

La classe est évidemment "thread-safe".

```
$ java NombresAtomiques
3
4
Somme de CalculComplique3
de 3 et 4 = 1215752199
```

Thread (17/38): partage thread-safe de variable : Synchronisateur ….

TachesAvecBarriere.java
CalculComplique4.java

CountDownLatch

```
public class CalculComplique4 implements Runnable {
  private int x;
  private AtomicInteger result;
  private CountDownLatch barriere;
  public CalculComplique4(int x, AtomicInteger r, CountDownLatch b) {
    this.x=x; this.result=r; barriere=b;
  }
  public void run() {
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
     result.addAndGet(y);
     barriere.countDown();
  }
}
```

Les objets (donc classe) synchronisateurs régulent le flux des threads.
CountDownLatch est un synchonisateur de type loquet à décrémentation :
le loquet est "armé" avec une certaine valeur entière,
il est décrémenté par la méthode countDown() ,
la méthode await() met en attente le thread jusqu'à ce que le loquet soit à zéro.

```
public class TachesAvecBarriere {
  ...
   int nombre = scanIn.nextInt();
   AtomicInteger resultat = new AtomicInteger(0);
   CountDownLatch barriere = new CountDownLatch(nombre);
   for (int i=1; i<=nombre; i++) {
     Thread tache = new Thread(new CalculComplique4(nombre, resultat, barriere));
     tache.start();
   }
   barriere.await();
   System.out.println("Somme CalculComplique4 de 1 à "+nombre+" = "+resultat.intValue());
```

```
$ java TachesAvecBarriere
5
Somme de CalculComplique4
de 1 à 5 = 891896857
```

Thread (18/38)

les synchronisateurs

De base :
volatile "synchronise" la valeur de la variable,
synchronized(objet) est un verrou à un seul accès
wait() et notify() et notifyAll() sont, en complément de synchronized, un dispositif blocage/déblocage.

Dans le package java.util.concurrent :
AtomicNumber permet des opérations "atomiques thread-safe"
CountDownLatch est une barrière à 1 point/moment de synchronisation,
CyclicBarrier une barrière posant plusieurs points de synchronisation
Semaphore
ReentrantLock généralise les verrous avec des conditions de verrouillage plus complexe , avec des méthodes "moins" bloquantes"
BlockingQueue et ConcurrentLinkedDeque proposent des techniques de "production-consommation"

TestArrayList.java
TestVector.java

Thread (19/38) : Collections et Multithreading : ArrayList Vector ...

```
import java.util.*;
public class TestArrayList {
  private ArrayList<String> maListe = new ArrayList<String>();
  public static void main (String [] args) throws InterruptedException {
    TestArrayList test = new TestArrayList();
    test.go();
  }
  public void go() throws InterruptedException {
    Thread tache1 = new Thread(new AccedeurListe(maListe));
    Thread tache2 = new Thread(new AccedeurListe(maListe));
    tache1.start();
    tache2.start();
    tache1.join();
    tache2.join();
    System.out.println("maListe.size() : "+maListe.size());
  }
  class AccedeurListe implements Runnable {
    private ArrayList<String> laListe;
    public AccedeurListe(ArrayList<String> l) {
      laListe = l;
    }
    public void run() {
      for (int i=0; i<200; ++i)
        laListe.add("abc");
    }
  }
}
```

```
$ java TestArrayList
maListe.size() : 400
$ java TestArrayList
maListe.size() : 400
…
$ java TestArrayList
maListe.size() : 398
$ java TestVector
monVector.size() : 400
$ java TestVector
monVector.size() : 400
...
```

Les Collections du package java.util ne sont pas "thread-safe", excepté Vector.

```
import java.util.*;
public class TestVector {
  private Vector<String> monVector = new Vector<String>();
  . . .
```

TestVector2.java

Thread (20/38) : Collections et Multithreading : Vector

```
public class TestVector2 {
  ...
  public static String getLast(Vector<String> vect) {
    int lastIndex = vect.size()-1;
    return vect.get(lastIndex);
  }
  public static void deleteLast(Vector<String> vect) {
    int lastIndex = vect.size()-1;
    vect.remove(lastIndex);
  }
  public void go() throws InterruptedException {
    Thread tache1 = new Thread(new AccedeurVector(monVector));
    Thread tache2 = new Thread(new AccedeurVector(monVector));
    for (int i=0; i<4000; ++i)
      monVector.add(i+"-ième ");
    tache1.start();
    tache2.start();
    tache1.join();
    tache2.join();
  }
  class AccedeurVector implements Runnable {
    private Vector<String> leVector;
    public AccedeurVector(Vector<String> l) {leVector = l;}
    public void run() {
      for (int i=0; i<1000; ++i) {
        System.out.println("monVector.getLast() : "
                           +getLast(monVector));
        deleteLast(monVector);
      }
    }
  }
```

```
$ java TestVector2
monVector.getLast() :
3999-ième
...
monVector.getLast() :
2000-ième
$ java TestVector2
monVector.getLast() :
3999-ième
...
monVector.getLast() :
2465-ième
$
```

La classe Vector est "thread-safe" donc (presque) chacune de ses méthodes.

Ça ne garantie pas que la composition d'appel à ses méthodes soit "thread-safe".

Obtenir le dernier et supprimer le dernier ne sont pas des actions irréalistes !

Thread (21/38) : Collections et Multithreading : Vector

TestVector3.java

```
public class TestVector3 {
  private Vector<String> monVector = new Vector<String>();
  public static String getLast(Vector<String> vect) {
    synchronized(vect) {
      int lastIndex = vect.size()-1;
      return vect.get(lastIndex);
    }
  }
  public static void deleteLast(Vector<String> vect) {
    synchronized(vect) {
      int lastIndex = vect.size()-1;
      vect.remove(lastIndex);
    }
  }
. . .
```

```
$ java TestVector3
monVector.getLast() :
3999-ième
...
monVector.getLast() :
2000-ième
$
```

Le verrouillage par synchronized offre une solution.
Attention à ne pas en abuser sur des temps longs, sinon ça ne sert à rien de paralléliser.
Mauvais exemple :

```
synchronized(vecteur) {
  for (int i=0; i<vecteur.size(); ++i)
    méthodeAFaire(vecteur.get(i));
}
```

Si possible, un meilleur exemple :

Le risque est qu'un élément enlevé du vecteur original soit impacté par la "méthodeAFaire".

```
Vector copieVecteur = null;
synchronized(vecteur) {
  CopieVecteur = (Vector) vecteur.clone();
}
for (int i=0; i<copieVecteur.size(); ++i)
  méthodeAFaire(copieVecteur.get(i));
}
```

Thread (22/38) : Collections et Multithreading : Iterator

TestVector4.java

```
public class TestVector4 {
  private Vector<String> monVector = new Vector<String>();
  public static void main (String [] args) throws InterruptedException {
    TestVector4 test = new TestVector4();
    test.go();
  }
  public void go() throws InterruptedException {
    Thread tache = new Thread(new RemoveurVector(monVector));
    for (int i=0; i<4000; ++i)
      monVector.add(i+"-ième ");
    tache.start();
    Iterator parcoursMonVector = monVector.iterator();
    while (parcoursMonVector.hasNext())
      System.out.println("parcoursMonVector.next() = "+parcoursMonVector.next());
    tache.join();
  }
  class RemoveurVector implements Runnable {
    private Vector<String> leVector;
    public RemoveurVector(Vector<String> l) {
      leVector = l;
    }
    public void run() {
      for (int i=0; i<100; ++i) {
        leVector.remove((int)(Math.random()
                        *leVector.size()));
      }
    }
  }
}
```

La documentation de Vector précise que l'iterator fourni est "fail-fast" : une ConcurrentModificationException est levée quand un thread remove un élément de la collection.

Pour être "thread-safe", try-catcher l'exception bien que ce ne soit pas obligatoire ou changer de stratégie de programmation.

```
$ java TestVector4
...
  parcoursMonVector.next() = 109-ième
Exception in thread "main"
     java.util.ConcurrentModificationException
    At ...
    at TestVector4.go(TestVector4.java:15)
    at TestVector4.main(TestVector4.java:6)
```

Thread (23/38) : Collections et Multithreading : CopyOnWriteArrayList du package java.util.concurrent

TestCopyOnWriteArrayList.java

```
import java.util.concurrent.*;
public class TestCopyOnWriteArrayList {
  private CopyOnWriteArrayList<String> maListe = new CopyOnWriteArrayList<String>();
  . . .
  public void go() throws InterruptedException {
    Thread tache = new Thread(new RemoveurCopyOnWriteArrayList(maListe));
    for (int i=0; i<4000; ++i)
      maListe.add(i+"-ième ");
    Iterator parcoursListe = maListe.iterator();
    tache.start();
    while (parcoursListe.hasNext())
      System.out.println("parcoursListe.next() = "+parcoursListe.next());
    tache.join();
  }
  class RemoveurCopyOnWriteArrayList implements Runnable {
    private CopyOnWriteArrayList<String> laListe;
    public RemoveurCopyOnWriteArrayList(CopyOnWriteArrayList<String> l) {
      laListe = l;
    }
    public void run() {
      for (int i=0; i<100; ++i) {
        laListe.remove((int)(Math.random()*laListe.size()));
      }
    }
  }
}
```

Le package java.util.concurrent fournit des classes pour la programmation multithread.
La classe CopyOnWriteArrayList est "totalement" "thread-safe" :
ses méthodes sont fort coûteuse en temps d'exécution.
Sa méthode iterator() réalise une copie de la liste comme précédemment suggéré.

Thread (24/38)
L'ArrayBlockingQueue synchronisateur et le Pattern Producteur-Consommateur

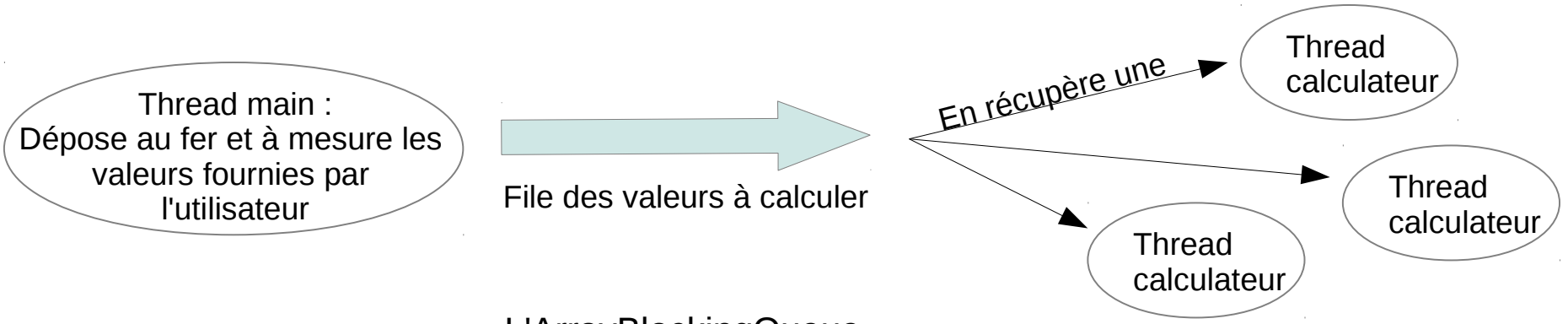
ProductionConsommation.java
CalculComplique5.java

le pattern producteur-consommateur

Le(s) producteurs produisent,
Déposent leur produit dans la file d'attente,
Attendent si elle est pleine

File de produit

Le(s) consommateurs consomment les produits,
Les prennent dans la file d'attente,
Attendent si elle est vide



put() y dépose un élément ou attend si la file est pleine.

L'ArrayBlockingQueue collection « file » thread-safe
synchroniseur conçu selon le pattern producteur-consommateur

take( ) récupère un élément ou attend si la file est vide.

Thread (25/38)

ProductionConsommation.java
CalculComplique5.java

L'ArrayBlockingQueue synchronisateur et le Pattern Producteur-Consommateur

```
public class ProductionConsommation {
  public static void main (String [] args) throws InterruptedException {
    ArrayBlockingQueue<Integer> fileDeJob = new ArrayBlockingQueue<Integer>(5);
    for (int i=0; i<=3; i++) {
      Thread tache = new Thread(new CalculComplique5(fileDeJob));
      tache.start();
    }
    Scanner scanIn = new Scanner(System.in);
    while (scanIn.hasNextInt())
      fileDeJob.put(scanIn.nextInt());
...
```

```
public class CalculComplique5 implements Runnable {
  private ArrayBlockingQueue<Integer> fileDeJob;
  public CalculComplique5(ArrayBlockingQueue<Integer> f) {
    fileDeJob = f; }
  public void run() {
    try {
      while (true) {
        int y = fileDeJob.take();
        int x = y;
        for (int i=0; i< 100000000; i++)
          for (int j=0; j< 100000000; j++)
            y +=  5;
         System.out.println("CalculComplique5("+x+")="+y);
      }
    } catch (InterruptedException ie) {}
  }
}
```

```
$ java ProductionConsommation
4 5 6 7 8
CalculComplique5(4) = 784662532
CalculComplique5(6) = 784662534
CalculComplique5(5) = 784662533
CalculComplique5(7) = 784662535
9
CalculComplique5(8) = 784662536
CalculComplique5(9) = 784662537
^C
```

ExecutionSequentielle.java
CalculComplique.java

Thread (26/38) : comment exécuter des taches ?
Exécution séquentielle

```
public class CalculComplique implements Runnable {
  private int x;
  public CalculComplique(int x) { this.x = x; }
  public void run() {
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
     System.out.println("CalculComplique("+x+") = "+y);
  }
}
```

```
public class ExecutionSequentielle {
  public static void main (String [] args) {
    Scanner scanIn = new Scanner(System.in);
    int nombre = 0;
    while (scanIn.hasNextInt()) {
      nombre = scanIn.nextInt();
      CalculComplique tache = new CalculComplique(nombre);
      tache.run();
    }
  }
}
```

```
$ java ExecutionSequentielle
1
CalculComplique(1) = -1539607551
4
CalculComplique(4) = -1539607548
q
$
```

Une tâche a besoin d'un thread pour être exécutée.

Dans un programme strictement séquentiel, c'est le thread main qui exécute les tâches les unes après des autres.

Solution valable car la tâche CalculCompliquee ne fait que des calculs et n'est jamais en attente d'E/s ou de ressources.
Sinon, d'autres threads pourraient s'exécuter sur le(s) processeur(s) pendant les attentes et blocages.

UnThreadParTache.java
CalculComplique.java

Thread (27/38) : comment exécuter des taches ?
Exécution parallèle "fanatique"

```
$ java UnThreadParTache
3
4
CalculComplique(3) = -1539607549
CalculComplique(4) = -1539607548
q
$
```

```
public class UnThreadParTache {
  public static void main (String [] args) {
    Scanner scanIn = new Scanner(System.in);
    int nombre = 0;
    while (scanIn.hasNextInt()) {
      nombre = scanIn.nextInt();
      Thread tache = new Thread(new CalculComplique(nombre));
      tache.start();
    }
  }
}
```

Solution valable dans le cas d'une charge modérée et de tâches faisant des E/S ou des attentes sur ressources : exemple serveur web.
Catastrophe si la charge augmente : temps de changement de contexte de thread, consommation mémoire …

```
public class CalculComplique implements Runnable {
  private int x;
  public CalculComplique(int x) { this.x = x; }
  public void run() {
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
     System.out.println("CalculComplique("+x+") = "+y);
  }
}
```

Runnable s'avère pauvre pour implémenter les tâches :
pas de résultat à retourner (directement),
pas d'Exception à propager si la tâche échoue,
pas d'information sur l'état d'avancement de la tâche.

Thread (28/38) : comment exécuter des taches ?
Executor … de tâches

ExecuteurDeTaches.java
CalculCompliquee.java

```
import java.util.concurrent.*;
public class ExecuteurDeTaches {
  public static void main (String [] args) {
    ExecutorService exec = Executors.newFixedThreadPool(2);
    Scanner scanIn = new Scanner(System.in);
    int nombre = 0;
    while (scanIn.hasNextInt()) {
      nombre = scanIn.nextInt();
      CalculComplique tache = new CalculComplique(nombre);
      exec.submit(tache);
    }
    exec.shutdown();
  }
}
```

Dans le package java.util.concurrent, autour de l'interface Executor, plusieurs classes permettent de:
- gérer la soumission des tâches
- gérer le cycle de vie des tâches.

La classe Executors est une fabrique de pool de threads pour exécuter les tâches.
La stratégie newFixedThreadPool crée un pool de threads de taille fixe.
Il existe d'autres stratégies ...

La classe ExecutorService :
La méthode submit(Runnable) soumet une tâche à l'exécuteur qui le fera … ultérieurement selon sa stratégie.
La méthode shutdown() demande à l'exécuteur de se terminer quand toutes ses tâches déjà soumises seront terminées.
La méthode shutdownNow() arrête l'exécuteur et tente d'annuler les tâches en cours.
....

```
$ java ExecuteurDeTaches
3
4
5
CalculComplique(3) = -1539607549
CalculComplique(4) = -1539607548
CalculComplique(5) = -1539607547
q
$
```

Thread (29/38) : comment exécuter des taches ?
Future

```
$ java MaitriseDesTaches
3
4
Somme de CalculComplique2 de 3
et 4 = 1215752198
```

```
import java.util.concurrent.*;
public class MaitriseDesTaches {
  public static void main (String [] args)
         throws InterruptedException, ExecutionException {
    ExecutorService exec = Executors.newSingleThreadExecutor();
    Scanner scanIn = new Scanner(System.in);
    int nombre1 = scanIn.nextInt();
    CalculComplique2 tache1 = new CalculComplique2(nombre1);
    Future<Integer> future1 = exec.submit(tache1);
    int nombre2 = scanIn.nextInt();
    CalculComplique2 tache2 = new CalculComplique2(nombre2);
    Future<Integer> future2 = exec.submit(tache2);
    int result = future1.get().intValue() + future1.get().intValue();
    System.out.println("Somme de CalculComplique2 de "+nombre1
                                        +" et "+nombre2+" = "+result);
    exec.shutdown();
  }
}
```

NewSingleThreadExecutor() correspond à la stratégie séquentielle de la classe Executors.

La méthode submit() de la classe ExecutorService renvoie un objet Future à la soumission d'une tâche.

La classe Future permet de suivre le cycle de vie d'une tâche : son état (attente, en exécution, terminée, interrompue), son résultat s'il y a.

Calculcompliquee2.java
MaitriseDesTaches.java

Thread (30/38) : comment exécuter des taches ? Callable et Future

```
import java.util.concurrent.*;
public class CalculComplique2 implements Callable<Integer>
{
  private int x;
  public CalculComplique2(int x) { this.x = x; }
  public Integer call() {
    int y = x;
    for (int i=0; i< 100000; i++)
      for (int j=0; j< 100000; j++)
        y +=  5;
    return y;
  }
}
```

L'interface Callable décrit une tâche fournissant un résultat ou propageant une exception par sa méthode call().

La classe Future possède :
La méthode get() pour récupérer de résultat de la tache, et attend si elle n'est pas terminée. Elle peut lancer l'InterruptedException et l'ExecutionException si la tâche "crash".
La méthode cancel() pour tenter d'arrêter la tache.
La méthode isDone() pour savoir si la tache est finie sans probléme, et sinon isCancelled().

Thread (31/38)

GUIetTacheLongue1.java

Thread et Application Graphique

```
public class GUIetTacheLongue1 extends JFrame {
  JProgressBar barreProgression;
  public static void main(String[] args) {
    new GUIetTacheLongue1();   }
  public GUIetTacheLongue1() {
    …
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    barreProgression = new JProgressBar();
    barreProgression.setMinimum(0);
    barreProgression.setMaximum(99);
    barreProgression.setValue(0);
    JButton bouton = new JButton("Demarrer la tache longue");
    bouton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          for(int i = 0; i < 100; i++ ) {
            barreProgression.setValue(i);
            System.out.print(".");
            try {
              Thread.sleep(100);
            } catch (InterruptedException ie) {}
          }
          System.out.println("") ;
        }
      }
    );
    pack();
    setVisible(true);
  }
}
```

```
$ java GUIetTacheLongue1
```

Un bouton permet de déclencher la tache longue.
Le traçage en console indique bien la progression mais pas en GUI.
La progression ne s'affiche en GUI que quand elle est finie !
De plus, l'arrêt en cliquant sur le X de la frame ne fonctionne pas.
Par contre, le <ctrl>C en console fonctionne.

GUIetTacheLongue2.java

Thread et Application Graphique

```
public class GUIetTacheLongue2 extends JFrame {
  ...
    JButton bouton = new JButton("Demarrer la tache longue");
    bouton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          Thread tache = new Thread(new LongueTache2(barreProgression));
          tache.start();
        }
      }
    );
    pack();
    setVisible(true);
  }
}
```

Le traitement de l'événement consiste à lancer un autre thread de traitement de la longue tache.
Cela semble bien fonctionner ….
En fait, il y a un risque d'interblocage !

```
class LongueTache2 implements Runnable {
  private JProgressBar barreProgression;
  public LongueTache2(JProgressBar bp) { barreProgression=bp; }
  public void run() {
    for(int i = 0; i < 100; i++ ) {
      barreProgression.setValue(i);
      System.out.print(".");
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {}
    }
    System.out.println("") ;
  }
}
```

```
$ java GUIetTacheLongue2
```

……..

……..

ThreadSwing.java

Thread de traitement des événements

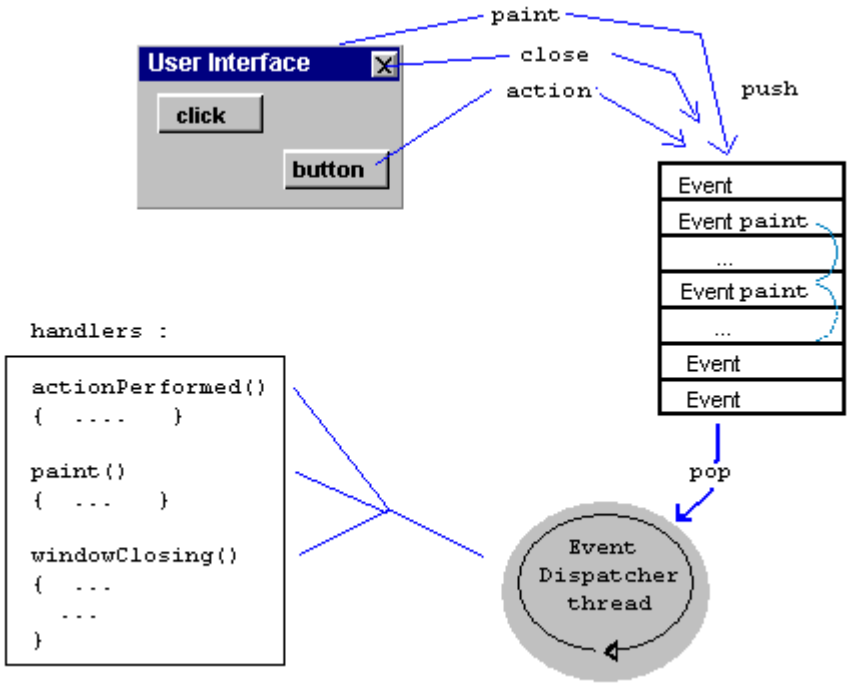
```
class ThreadSwing extends JFrame {
  public static void main(String args[]) {
    System.out.println("main -----> thread name = "
                       + Thread.currentThread().getName());
    new ThreadSwing();
  }
  public ThreadSwing() {
    this.setTitle("ThreadSwing");;
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JButton bouton = new JButton("bouton");
    this.getContentPane().add(bouton);
    bouton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.out.println("actionPerformed ---> thread name = "
                             + Thread.currentThread().getName());
        }
      });
    this.pack();
    this.setVisible(true);
  }
  public void paint(Graphics g) {
    super.paint(g);
    System.out.println("paint -----> thread name = "
                       + Thread.currentThread().getName());
  }
}
```

2 threads pour l'application :
Celui du main
Celui qui traite l'événement et le paint()

```
$ java ThreadSwing
main -----> thread name = main
paint -----> thread name = AWT-EventQueue-0
paint -----> thread name = AWT-EventQueue-0
actionPerformed -----> thread name = AWT-EventQueue-0
```

Thread (34/38)

thread Event Dispatcher



Tous les "frameworks" graphiques sont (malheureusement) monothread à cause :
De la complexité des bibliothèques
De la sensibilité aux interblocages, notamment du au MVC qui laisse une grande liberté/souplesse au programmeur
Pour Swing, du fait que les composants graphiques ne sont pas thread-safe.

Globalement, la thread-safety d'un GUI est basé sur le traitement monothread des événements, des rafraîchissements (paint) et des modifications des composants.

Conséquences :
Le programmeur doit s'y plier pour tendre vers la thread-safety.
C'est le pattern d'unicité de traitement graphique.
La bibliothèque swing offre des outils pour ce faire.

Thread (35/38)

GUIetTacheLongue3.java

SwingUtilities.invokeLater()

```
public class GUIetTacheLongue3 extends JFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater( new Runnable() {
          public void run() { new GUIetTacheLongue3(); }
    });
  }
...

class LongueTache3 implements Runnable {
  private JProgressBar barreProgression;
  public LongueTache3(JProgressBar bp) { barreProgression=bp; }
  public void run() {
    for(int i = 0; i < 100; i++ ) {
      final int niveau = i;
      SwingUtilities.invokeLater( new Runnable() {
          public void run() { barreProgression.setValue(niveau); }
      });
      System.out.print(".");
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {}
    }
    System.out.println("") ;
  }
}
```

La méthode invokeLater( ) de SwingUtilities planifie une tache qui s'exécutera dans le thread de traitement des événements.
C'est une sorte d'Executor.

SwingUtilities possède d'autres méthodes.

```
$ java GUIetTacheLongue3
```


```
.........
```


```
.........
```

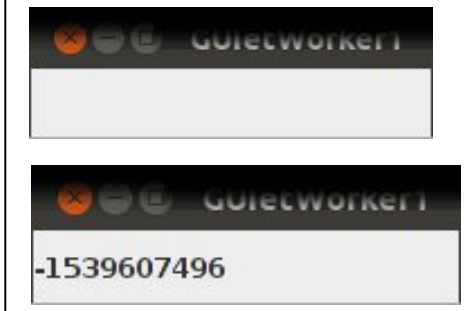
En renvoyant la création du GUI du thread main au thread EventDispatcher et en renvoyant la modification de la JProgressBar du thread "longue tache" au thread EventDispatcher, le pattern d'unicité de traitement graphique est respecté.

Thread (36/38)
SwingWorker

GUIetWorker1.java

```
public class GUIetWorker1 extends JFrame {
  public GUIetWorker1() {
    ...
    final JLabel resultat = new JLabel("                    ");
    class CalculComplique6 extends SwingWorker<Integer, Void> {
      private int x;
      public CalculComplique6(int x) { this.x=x; }
      public Integer doInBackground() {
        int y = x;
        for (int i=0; i< 100000; i++)
          for (int j=0; j< 100000; j++)
        y +=  5;
        return y;
      }
      protected void done() {
        try {
          resultat.setText(get().toString());
         } catch (InterruptedException ie) {
         } catch (ExecutionException ee) {}
      }
    }
    this.getContentPane().add(resultat);
    this.pack();
    this.setVisible(true);
    CalculComplique6 calcul = new CalculComplique6(56);
    calcul.execute();
  }
}
```

```
$ java GUIetWorker1
```



La classe SwingWorker<T,V> réalise les workers threads ou background threads. Elle implémente Future.

T est le type du résultat de la tache et V le type des résultats intermédiaires s'il y a.

Sa tache est effectuée par la méthode doInBackground() qui retourne un résultat.
Sa méthode done() est automatiquement lancée dans le thread EventDispatcher à la fin de la tache.

Sa méthode get() retourne le résultat et sinon l'attend. Elle lève les exceptions "classiques".

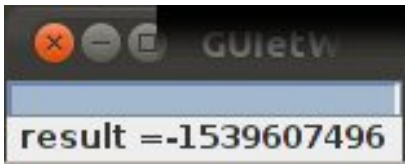
Thread (37/38)

GUIetWorker2.java

SwingWorker progression

```
...
  final JProgressBar progressBar = new JProgressBar(0, 100);
  class CalculComplique7 extends SwingWorker<Integer, Void> {
    private int x;
    public CalculComplique7(int x) { this.x=x; }
    public Integer doInBackground() {
      int y = x;
      for (int i=0; i< 100000; i++) {
        for (int j=0; j< 100000; j++)
          y +=  5;
        setProgress(i/1000);
      }
      return y;
    }
    protected void done() {
      try {
        labelResult.setText("result ="+get().toString());
       } catch (InterruptedException ie) {
       } catch (ExecutionException ee) {}
    }
  }
  CalculComplique7 calcul = new CalculComplique7(56);
  calcul.addPropertyChangeListener(
   new PropertyChangeListener() {
       public  void propertyChange(PropertyChangeEvent evt) {
           if ("progress".equals(evt.getPropertyName())) {
               progressBar.setValue((Integer)evt.getNewValue());
           }
       }
   });
  calcul.execute();
```

```
$ java GUIetWorker2
```



SwingWorker intègre des dispositifs de progression de la tache en background.
La méthode setProgress( ) affecte la propriété "liée" (bound property) : ce qui permet d'écouter ses changements en s'inscrivant comme PropertyChangeListener de la tache worker.

Thread (38/38)

GUIetWorker3.java

SwingWorker annulation

```
...
  boite.add(boutonAnnule);
  class CalculComplique7 extends SwingWorker<Integer, Void> {
    ...
    public Integer doInBackground() {
      int y = x;
      for (int i=0; (i<100000) && !isCancelled(); i++) {
        for (int j=0; (j<100000) && !isCancelled(); j++)
          y +=  5;
        setProgress(i/1000);
      }
      return y;
    }
    protected void done() {
      try {
        if (!isCancelled())
          labelResult.setText("result ="+get().toString());
       } catch (InterruptedException ie) {
       } catch (ExecutionException ee) {}
    }
  }
  ...
  boutonAnnule.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        if (!calcul.isDone()) {
          calcul.cancel(true);
          labelResult.setText("annulé");
        }
      }
  });
```

```
$ java GUIetWorker3
```

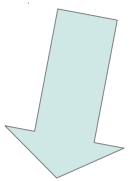
SwingWorker intègre un dispositif d'annulation de la tache en background.

La méthode isDone( ) retourne true si la tache est accomplie.
La méthode cancel(force) tente d'annuler la tache : si force est true, alors l'annulation se fait même si la tache est en cours d'exécution.
La méthode isCancelled( ) retourne true si la tache est annulée.

Thread (39/38)

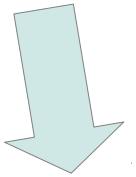
Quoi de neuf dans JAVA 8 ?

AtomicInteger résolvait le problème des accès concurrents sur un compteur numérique :
La méthode addAndGet() bloque le thread aussi longtemps que d'autres threads utilisent l'AtomicInteger … perte de temps



Les LongAdders fonctionnent ainsi :
La méthode add() ajoute directement la valeur si le « londAdder » n'est pas « pris » par un autre thread,
sinon le delta est mémorisé et sera ajouté quand le « londAdder » sera « libre ».

Le verrou ReadWriteLock permettait de diviser le programme en blocs qui doivent être en exclusion mutuelle (écritures), et en blocs qui n'en ont pas besoin (lectures).
Mais il était très lent à l'exécution.



Le StampedLock fonctionne sur 2 modes selon un pari :
La lecture se fait en acquérant le verrou en mode « optimiste ». Le déverrouillage est valide s'il n'y a pas eu d'écriture depuis : mode rapide si le pari est ok !
Sinon il faut alors passer en mode « pessimiste » de verrouillage classique.

Gain de temps, code plus rapide mais programmation plus exigeante