Java

IUP 2 - POO 3

Plan du cours

- Rappels
- Exceptions
- Classes abstraites et interfaces
- Polymorphisme et liaison différée
- Collections : tableaux et structures dynamiques
- Threads et synchronisation
- Entrées-Sorties

Rappels

Rappels

- types natifs de tailles fixes avec classes enveloppes immuables :

Types	Valeurs	Tailles	Enveloppes
boolean	true/false	1	Boolean
char	Unicode	16	Character
byte	Entier signé	8	Byte
short	Entier signé	16	Short
int	Entier signé	32	Integer
long	Entier signé	64	Long
float	Flottant IEEE 754	32	Float
double	Flottant IEEE 754	64	Double

Rappels

- Voir aussi : http://java.sun.com/docs/books/tutorial/
- Une déclaration d'objet crée une référence.
  - Cette référence désignera l'instance créée par l'opération `new`.
  - L'opérateur `new` s'applique à un constructeur d'une classe pour obtenir une instance.
  - Les objets sont créés sur le tas, et nettoyés par un ramasse-miettes.
  - Il n'y a pas de destructeur d'instance

Tableaux

- Syntaxe :

```
typeElements nomTableau[]; // Syntaxe à la C
```

ou :

```
typeElements[] nomTableau; // Syntaxe Java
```

- Une déclaration de tableau crée une référence, initialisée à null.
- Le tableau est créé dynamiquement par un appel à new : `nomTableau = new typeElements[taille];` (taille n'est pas nécessairement connue à la compilation).
- On peut également l'initialiser à sa déclaration :

```
int[] tab = {10, 9, 8, 7};
```

- Tout tableau dispose de l'attribut length.

Rappels

- Toute classe Java hérite de la classe Object.
- Il n'y a pas d'héritage multiple pour les classes (mais pour les interfaces).
- Syntaxe de l'héritage :

```
class UneClasse extends BaseClasse {
...
}
```

- Membres privés, publics, protégés, à visibilité de paquetage.

Exceptions

Exceptions

- Signale un état inhabituel ou une erreur.
- Une exception qui est levée peut être capturée (traitée) ou propagée (traitement délégué au niveau supérieur).
- Traitement des erreurs séparé du code (bloc de traitement).
- Une exception est une instance d'une classe Exception (prédéfinie ou définie par le programmeur).

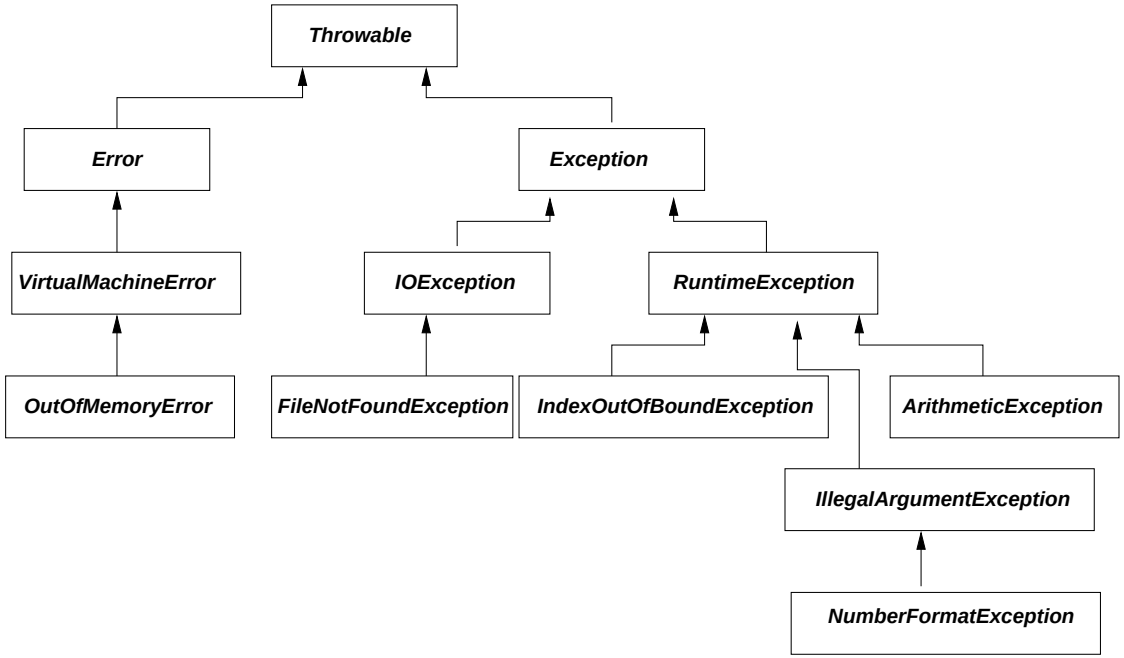
Exceptions

- Toute méthode doit préciser les exceptions qu'elle est susceptible de lever (celles qu'elles ne traitera pas).
- Le compilateur peut donc vérifier qu'elles sont bien gérées dans le programme.
- Certaines exceptions ne sont pas capturables (pas la peine de les préciser).
- On peut ignorer certaines exceptions, mais il faut l'indiquer explicitement.

Classes d'exceptions prédéfinies

- Toutes les exceptions sont des classes dérivant de java.lang.Exception ou de ses classes filles.
- Les paquetages définissent des classes filles spécialisées (java.io, par exemple, définit la classe IOException).
- L'API Java définit également java.lang.Error et ses classes filles pour les erreurs irrécupérables : on ne tente jamais de les capturer.

Exceptions


Throwable
Error
Exception
VirtualMachineError
IOException
RuntimeException
OutOfMemoryError
FileNotFoundException
IndexOutOfBoundException
ArithmeticException
IllegalArgumentException
NumberFormatException

Gestion des exceptions

- Instruction try/catch : les instructions levées dans la clause try sont traitées (ou non) dans les clauses catch.
- Les clauses catch sont évaluées dans l'ordre : la première qui correspond est choisie => il faut donc aller du plus spécifique au plus général !
- Une exception levée mais non traitée dans un catch est propagée à l'appelant... et ainsi de suite jusqu'à la méthode main() => interruption du programme.
- Le compilateur doit détecter quand une exception n'est jamais traitée.

Exemple

```
try {
  readFromFile(”monfic”);          // Peut lever une exception
  ...
}
catch(Exception e) {                  // Capturera donc tout type d’exception...
  System.err.println(”Exception “ + e + “ levée à la lecture du fichier”);
  ...
}

// Mieux :

try {
  readFromFile(”monfic”);       // Peut lever une exception
  ...
}
catch(FileNotFoundException e) {
  // Traitement de l’erreur “fichier non trouvé”
  ...
}
catch(IOException e) {
 // Traitement d’une erreur de lecture
 ...
}
catch(Exception e) {
 // Traitement de toutes les autres exceptions
 ...
}
```

Exceptions contrôlées/ non contrôlées

- Les méthodes doivent déclarer les exceptions qu'elles lèvent à l'aide de la clause throws :

```
void readFile(String s) throws IOException, InterruptedException {...}
```

- Toute méthode invoquant readFile() devra soit traiter les exceptions ainsi déclarées (avec try/ catch), soit les propager (en utilisant throws).

- Les filles de java.lang.RuntimeException ou de java.lang.Error ne sont pas contrôlées : il n'est pas nécessaire de les déclarer.

Levée d'exception

- Une exception peut être levée explicitement avec l'instruction `throw` appliquée à une instance d'exception.
- L'exécution s'arrête et le contrôle passe au bloc `try/catch` le plus proche.
- Une levée d'exception ne doit avoir lieu que dans les cas exceptionnels (pour les situations classiques, on utilisera les tests et les traitements classiques).

Exemple

```
if (...) throw new Exception();      // Exception de base...
if (...) throw new Exception("Un problème est survenu...");

// Cette méthode peut lever SecurityException, qui dérive de
// RuntimeException et n'est donc pas contrôlée...
// Il n'est donc pas nécessaire de la déclarer avec throws

public void checkRead(String s) {
  if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
    throw new SecurityException("Accès à " + s + " interdit");
}
```

Classes abstraites et interfaces

Classes abstraites

- Une méthode abstraite est déclarée par abstract et n'a pas de corps : c'est uniquement une signature :

```
abstract void méthodeAbstraite(String param);
```

- Une classe contenant une méthode abstraite est une classe abstraite : elle doit être déclarée abstract et ne pourra pas être instanciée.

```
abstract class ClasseAbstraite {
  ...
  abstract void méthodeAbstraite(String param);
  ...
}
```

Classes abstraites (suite)

- Une classe abstraite peut également contenir des méthodes non abstraites
- Une classe abstraite doit être dérivée et toutes ses méthodes abstraites doivent être redéfinies.
- Une classe fille qui ne redéfinit pas une méthode abstraite est elle-même une classe abstraite.

Interfaces

- Une interface définit un ensemble de méthodes que devra implémenter une classe.
- Une classe implémente une interface lorsqu'elle redéfinit toutes les méthodes de cette interface.
- Une interface ne contient que des signatures de méthodes abstraites et, éventuellement, des constantes.
- Le mot-clé `abstract` est implicite.

Interfaces (suite)

- Une classe peut implémenter plusieurs interfaces.
- Une interface est une "promesse", elle peut être utilisée comme un type de classe : une liste de "comparables" est une liste d'objets implémentant l'interface Comparable.
- Une interface peut être vide : elle sert alors à indiquer qu'une classe implémente une fonctionnalité particulière : Cloneable, Serializable, par exemple.
- Une interface peut dériver d'une ou plusieurs autres interfaces

Exemple

```
interface Conduisible {
  boolean demarrerMoteur();
  void arreterMoteur();
  float accelerer(float accel);
  boolean tourner(Direction vers);
}

class Automobile implements Conduisible {
 ...
 public boolean demarrerMoteur()  {
    moteurTourne = true;
    ...
 }
 public void arreterMoteur() {
   moteurTourne = false;
    ...
 }
 public float accelerer(float accel) { ... }
 public boolean tourner(Direction vers) { ... }
 public void klaxonner() { ... }  // Propre à une automobile
...
}
```

Exemple (suite)

```
class TondeuseGazon implements Conduisible {
  ...
  public boolean sacPlein() { ... } // Propre aux tondeuses
}

Automobile voiture = new Automobile();
Tondeuzegazon tondeuse = new TondeuseGazon();
Conduisible vehicule;

vehicule = voiture;
vehicule.demarrerMoteur();
vehicule.arreterMoteur();

vehicule = tondeuse;
vehicule.demarrerMoteur();
vehicule.arreterMoteur();
```

Polymorphisme et liaison différée

Type statique et type dynamique

- Tout objet Java a un type statique : celui qu'il a à la compilation. C'est le seul que le compilateur connaît.
- Un objet Java a également un type dynamique : celui qu'il a lors de l'exécution. Il peut être différent du type statique.
- Lorsqu'une méthode est appliquée à un objet, c'est celle de son type dynamique.
- Comme le compilateur ne connaît que le type statique, il ajoute du code aux méthodes redéfinies afin d'effectuer une liaison différée (late binding)

Appel polymorphe

- Soit le code :

```
Point p = new PointNommé(”Bla”, 10, 5);
System.out.println(p);
System.out.println(p.getNom()); // Erreur !
```

- À la compilation, il doit exister une méthode toString() et une méthode getNom() pour un objet de type Point ou d'une classe parente (sinon, erreur de compilation)

- À l'exécution, Java recherche la méthode toString() à appeler : c'est la dernière en allant du type statique (Point) vers le type dynamique (PointNommé)

Comparaison avec C++

- En C++, pour qu'un appel polymorphe soit possible, la méthode doit avoir été déclarée `virtual`. Sinon, par défaut, ce sera toujours celle du type statique qui sera appelée.
- En Java, c'est le contraire, il faut indiquer qu'une méthode est `final` si on ne veut pas qu'elle puisse être redéfinie : le compilateur n'ajoutera pas de code pour la liaison dynamique.
- De même, une classe `final` ne pourra être dérivée (toutes ses méthodes sont considérées comme `final`).

Classes et méthodes finales

- Efficacité : le traitement de la liaison dynamique est plus lourd que celui de la liaison statique. Une méthode `final` s'exécute plus rapidement (moins de code et possibilité d'expansion en ligne).

- Sécurité : la liaison dynamique ne permet pas de contrôler le code qui sera exécuté car on ne sait pas quelle méthode sera exécutée. Une méthode `final` offre cette garantie.

Tableaux et collections

Tableaux

- Collection linéaire de données de même type (internes ou instances de classe).
- Les éléments de types internes (`int`, `float`, ...) sont stockés dans le tableau.
- Dans le cas d'objets, ce sont les références qui sont stockées.
- PB: que faire quand le tableau est plein et qu'on veut lui ajouter d'autres éléments ?

Exemple

```
import java.util.*;
public class Tableau {
 public static void main(String[] argv) {
   int[] longueurMois1;                    // Déclaration d'une référence
   longueurMois1 = new int[12];            // Construction
   int[] longueurMois2 = new int[12];      // Forme abrégée
   int[] longueurMois3 = {                 //  Encore plus court et init
       31, 28, 31, 30, 31, 30,
       31, 31, 30, 31, 30, 31,
   };

   final int MAX = 10;
   int[][] moi = new int[MAX][];
   for (int i = 1; i < MAX; i++) {         // Tableau de 10 par 24
    moi[i] = new int[24];
   }

   // Tout tableau dispose d'un attribut 'length'
   System.out.println(moi.length);       // Affiche '10'
   System.out.println(moi[0].length);  // Affiche '24'
 }
}
```

Tableaux (suite)

- Lorsque l'on dépasse la taille maximale d'un tableau, on obtient une exception ArrayIndexOutOfBoundsException.
- Pour agrandir un tableau, on réalloue un nouveau tableau et on y transfère les données.
- Cette technique convient à des collections linéaires simples.
- Pour les structures plus complexes, on utilisera des collections dynamiques

Exemple

```
import java.util.*;
public class Tableau {
   private static int n;
   public static Calendar getDate() {
     if (n++ > 21) return null;
     return Calendar.getInstance();
   }
   public static void main(String[] args) {
    int nbDates = 0;
    final int MAX = 10;
    Calendar[] dates = new Calendar[MAX];
    while ((c = getDate() != null) {
      if (nbDates >= dates.length) {     // On réalloue MAX de plus...
        Calendar[] tmp = new Calendar[dates.length + MAX];
        System.arraycopy(dates, 0, tmp, 0, dates.length);
        dates = tmp;                              // Copie de la référence
                              // L'ancien tableau passera au ramasse-miettes...
      }  // if
      dates[nbDates++] = c;
    }  // while
    System.out.println("Taille du tableau = " + dates.length);
}
```

La classe Vector

- La classe Vector implémente une collection dynamique.
- L'instance de Vector croît automatiquement lorsque cela est nécessaire.
- Méthodes : add(o), add(i, o), clear(), contains(o), get(i), indexOf(o), remove(o), remove(i), size(), toArray(), etc. (voir API).
- Un Vector est un objet synchronisé (utilisable par plusieurs threads) => surcoût à l'exécution.

La classe ArrayList

- Identique à Vector, mais non synchronisée (plus rapide, mais nécessité de synchroniser quand plusieurs threads partagent une ArrayList).
- Constructeurs de Vector et d'ArrayList surchargés pour prendre une taille initiale (évite une réallocation).
- Dans la mesure du possible, préférer une ArrayList à un Vector.

Exemple

```
Vector v = new Vector();
Demo source = new Demo(15);

v.add(source.getDate());
v.add(source.getDate());
v.add(source.getDate());

System.out.println(”Récupération par l’indice :”);
for (int i = 0; i < v.size(); i++) {
  System.out.println(”Élément “ + i + “ = “ + v.get(i));
}

// Idem avec un ArrayList...
```

L'interface Collection

- Vector et ArrayList sont des implémentations particulières de l'interface List, qui est une implémentation de l'interface Collection pour représenter une collection ordonnée.
- L'interface Collection est l'ancêtre de toutes les collections, elle définit les opérations suivantes : add(o), remove(o), contains(o), size(), isEmpty(), iterator(), toArray().
- Une collection particulière peut lever UnsupportedOperationException si elle n'implémente pas add() ou remove()

L'interface Iterator

- Un itérateur permet de parcourir une suite de valeurs.
- L'interface Iterator ne possède que deux méthodes de parcours, next() et hasNext().
- L'interface fille, ListIterator, s'applique aux listes (implémentations de List) et permet de les parcourir dans les deux sens.
- Elle définit en plus hasPrevious(), previous(), nextIndex(), previousIndex().
- ListIterator permet aussi de modifier la liste pendant son parcours.

Exemple d'itérateur

```
public void printElements(Collection c, PrintStream out) {
  Iterator iterateur = c.iterator();

 // À partir de maintenant, on ignore le type de la Collection...
  while (iterateur.hasNext()) {
     out.println(iterateur.next());
  }
}
```

L'interface List

- List est une interface pour les collections ordonnées (les éléments ont des indices).
- Elle définit les opérations add(i, o), remove(i), get(i), set(i, o), indexOf(o), listIterator().
- L'interface List est implémentée par les classes ArrayList, LinkedList et Vector.
- Les instances de ces trois classes disposent donc des méthodes définies par Collection et List.
- Voir plus loin la classe Collections pour les méthodes statiques applicables aux List.

Exemple de liste

```
import java.util.*;

public class ListeChainee {
  public static void main(String[] args) {
    LinkedList liste = new LinkedList();
    liste.add(new Object());
    liste.add("Coucou");

    System.out.println("Liste des éléments");

    ListIterator iter = liste.listIterator();
    while (iter.hasNext())
      System.out.println(iter.next());

   if (liste.indexOf("Coucou") < 0)
      System.out.println("Coucou n'est pas dans la liste");
   else
      System.out.println("Coucou est dans la liste");
  }
}
```

L'interface Set

- Représente une collection où les éléments ne peuvent être dupliqués.
- N'ajoute aucune méthode par rapport à Collection, mais renforce le comportement de add().
- L'interface Set a une interface fille, SortedSet, qui ajoute les méthodes subset(o1, o2), headset(o), tailset(o), first(), last() et comparator().
- Set est implémentée par la classe HashSet et SortedSet par la classe TreeSet

Exemple

```
HashSet ens = new HashSet();

ens.add(“un”);
ens.add(”deux”);
ens.add(”un”);       // Duplication !
ens.add(”trois”);

Iterator iter = ens.iterator();
while (iter.hasNext())
    System.out.println(iter.next());
```

L'interface Map

- Permet d'implémenter des collections de paires clé/valeur (valeurs indicées par des clés).
- On peut donc indicer des éléments de type quelconque par des clés de types quelconques.
- Cette interface définit les opérations put(clé, val), get(clé), remove(clé), size(), keySet(), values().
- keySet() renvoie un Set de toutes les clés, values() renvoie une Collection de toutes les valeurs.

L'interface SortedMap

- Interface fille de Map, pour implémenter des Map triées sur les clés.
- Ajoute les méthodes subMap(clé1, clé2), headMap(clé), tailMap(clé), firstKey(), lastKey(), comparator().
- L'interface Map est implémentée par les classes HashMap, HashTable, LinkedHashMap et IdentityHashMap.
- L'interface SortedMap est implémentée par la classe TreeMap.

Exemple

```
HashMap societes = new HashMap();

societes.put(”Adobe”, “Mountain View, CA”);
societes.put(”Apple”, “Cupertino, CA”);
societes.put(”IBM”, “White Plains, NY”);
societes.put(”Sun”, “Mountain View, CA”);

String requete = “Apple”;
String resultat = (String)societes.get(requete);
System.out.println(requete + “ est située à “ + resultat);

Iterator iter = societes.keySet().iterator();
while (iter.hasNext()) {
  String cle = (String)iter.next();
  System.out.println(cle + “ : “ +  societes.get(cle));
}
```

Tris de collections

- Les classes Arrays et Collections fournissent des méthodes statiques pour trier des tableaux ou des collections.
- L'interface Comparator permet de définir un ordre de comparaison (par défaut, c'est l'ordre croissant) à l'aide des méthodes compare(o1, o2) et equals(o).
- Il suffit de définir une classe implémentant Comparator et rédéfinissant compare().

Exemples

```
public class TableauTrie {
  public static void main(String[] args) {
      String[] chaines = {”Bonjour”, “tout”, “le”, “monde”};

      Arrays.sort(chaines);
      for (int i = 0; i < chaines.length; i++)
         System.out.println(chaines[i]);
  }
}

public class CollectionTriee {
  public static void main(String[] args) {
      ArrayList chaines = new ArrayList();
      chaines.add(”Bonjour”);
      chaines.add(”tout”);
      chaines.add(”le”);
      chaines.add(”monde”);

     Collections.sort(chaines);
      for (int i = 0; i < chaines.size(); i++)
         System.out.println(chaines.get(i));
  }
}
```

Exemples (suite)

```
public class CompareLongueur implements Comparator {
   public int compare(Object o1, Object o2) {
     return o1.length() - o2.length();
   }
}

// Idem précédemment, mais avec les appels suivants :

Arrays.sort(chaines, new CompareLongueur());

Collections.sort(chaines, new CompareLongueur());
```

Recherches

binarySearch()	Dichotomique	Arrays, Collections
contains()	Linéaire	ArrayList, Vector, HashSet, HashTable, LinkedList
containsKey()		HashMap, HashTable, Properties, TreeMap
containsValue()		HashMap, HashTable, Properties, TreeMap
indexOf()		ArrayList, LinkedList, List, Stack, Vector
search()	Linéaire	Stack

Conversion en tableau

- Pour convertir n'importe quelle Collection en tableau, on utilisera la méthode toArray().
- Sans paramètre, elle renvoie un Object[]
- Avec un paramètre tableau, celui-ci sert à déterminer le type du tableau renvoyé.
- S'il est assez grand, c'est lui qui est rempli et renvoyé, sinon toArray() renvoie un nouveau tableau.
- Si le tableau résultat n'est pas un Object[], la Collection doit contenir des éléments de même type.

Exemple

```
import java.util.*;

public class VersTab {
  public static void main(String[] args) {
    ArrayList liste = new ArrayList();
    liste.add(”Bonobo”);
    liste.add(”Mandrill”);
    liste.add(”Babouin”);
    liste.add(new Date());

    // Conversion en Object[] : peut stocker des elts de tous types
    Object[] tabObjets = liste.toArray();
    System.out.println(”Taille du tableau = “ + tabObjets.length);

   // Conversion en String[] : ArrayStoreException à cause de
   // l’élément de type Date...
   // Noter l’utilisation d’un paramètre tableau vide,
   // uniquement pour forcer le type du tableau résultat
   String[] tabChaines = liste.toArray(new String[0]);
  }
}
```

Création d'itérateurs

- Pour implémenter un itérateur sur une structure de données SD que l'on a créée, il faut implémenter l'interface `Iterator`.
- La classe SD doit alors fournir au moins les méthodes `hasNext()` et `next()`.
- On peut également créer une classe itérateur distincte de la classe SD (ou être une classe interne de celle-ci). La classe SD fournira alors une méthode renvoyant un tel itérateur.

Exemple

```
import java.util.*;

public class DemoIterateur implements Iterator {

   protected String[] donnees = {"un", "deux", "trois"};

   protected int indice = 0;

   public boolean hasNext() {
    return (indice < donnees.length);
   }

   public Object next() {
    if (indice >= donnees.length)
       throw new IndexOutOfBoundException("....");
   return donnees[indice++];
  }

  public void remove() {
   throw new UnsupportedOperationException("...");
  }

  public static void main(String[] args) {
     DemoIterateur iter = new DemoIterateur();
     while (iter.hasNext()) System.out.println(iter.next());
  }
}
```

Structures à plusieurs dimensions

- Pour disposer de deux dimensions, il suffit d'utiliser un tableau ou une collection dont chaque élément sera un tableau ou une collection.
- Ce principe est généralisable à n dimensions
- Les dimensions n'auront pas nécessairement la même longueur...

La classe Properties

- Hachage spécialisé de chaînes pour mémoriser l'environnement d'exécution d'un programme.
- On manipule les propriétés avec setProperty(chaine, valeur), getProperty(chaine), save(fluxOut, titre), load(fluxIn).
- La classe java.lang.System fournit des informations sur le système à l'aide de la méthode statique System.getProperty().
- Une application peut définir des propriétés système à l'aide de System.setProperty() ou avec l'option -D de l'interpréteur java

Exemple

```
import java.util.*;
import java.io;

String monRep = System.getProperty(”user.home”);
Properties propSys = System.getProperties();

propSys.list(System.out);

Properties options = new Properties();
File fichierConfig = new File(monRep, “.config”);
try {
  options.load(new FileInputStream(fichierConfig));    // Chargement de la config
} catch (IOException e) { ... }

String couleur = options.getProperty(”color”, “gray”);  // gray sera pris par défaut

options.setProperty(”color”, “green”);

try {
  options.save(new FileOutputStream(fichierConfig), “Ma config à moi”);
} catch (IOException e) { ... }
```

Threads et synchronisation

Threads

- Première méthode : créer une classe héritant de java.lang.Thread, redéfinir sa méthode run() et créer une instance de cette nouvelle classe.
- Deuxième méthode : créer une classe implémentant l'interface runnable, donc définir une méthode run(), et passer une instance de cette classe au constructeur Thread().
- Dans les deux cas, l'objet est un Thread qui exécutera le code défini par la méthode run().
- Par défaut, Java alloue 1Mo de mémoire virtuelle par thread.

Threads

- Un thread est lancé par la méthode start() de Thread. Il exécute alors le code de run().
- Pendant l'exécution d'un thread, le thread initial poursuit son exécution avec l'instruction suivant l'appel à start().
- Ne jamais appeler directement run() : cela ne créerait pas de thread mais exécuterait uniquement le code de cette méthode.
- Après avoir exécuté run(), un thread se termine et disparaît.

Exemples

```
// Première méthode
class TriEnBackground extends Thread {
  ArrayList _liste;
  public TriEnBackground(ArrayList l) { _liste = l; }
  public void run() { Collections.sort(_liste); }
}

ArrayList maListe;
...
Thread tri = new TriEnBackground(maListe);
tri.start();

// Deuxième méthode (préférable)
class TriEnBackground implements Runnable {
  ArrayList _liste;
  public TriEnBackground(ArrayList l) { _liste = l; }
  public void run() { Collections.sort(_liste); }
}

ArrayList maListe;
...
Thread tri = new Thread(new TriEnBackground(maListe));
tri.start();
```

Exemples

- On peut également utiliser une instance anonyme de Runnable :

```
ArrayList maListe;
...
new Thread(new Runnable() {
        public void run() { Collections.sort(maListe); }
                        }).start();
```

- L'intérêt de la deuxième méthode est qu'elle permet en même temps d'hériter d'une autre classe.

Priorités et ordonnancement

- On peut utiliser des priorités différentes : un thread de priorité `p` ne s'exécutera que s'il n'y a pas de thread de priorité supérieure en attente.

```
t1.setPriority(Thread.NORM_PRIORITY - 1);
t2.setPriority(Thread.currentThread().getPriority() - 1);
```

- Java ne garantit pas que les threads de même priorité s'exécuteront à tour de rôle : la méthode de classe `Thread.yield()` met le thread courant en pause, ce qui permet de passer la main à un thread en attente.

Ordonnancement

- La méthode `Thread.sleep(n)` interrompt le thread courant avec `Thread.yield()`, attend `n` milli-secondes (sans relâcher les verrous) et reprend l'exécution. Avec cette méthode, il faut capturer `InterruptedException`.
- Selon les plateformes, le scheduling peut être préemptif ou non préemptif (Java lui-même n'est pas préemptif).

Exemple

```
class Patate implements Runnable {
 public void run() {
  while (true)
     System.out.println( Thread.currentThread().getName() );
 }
}

public class TestOrdonnancement {
  public static void main(String[] args) {
     Patate patate = new Patate();
     new Thread(patate, “Une patate”).start();
     new Thread(patate, “Deux patates”).start();
 }
}
```

Ordonnancement

- Si l'exécution de l'exemple précédent n'affiche que "Une patate", l'ordonnancement du système n'est pas préemptif.
- On peut forcer explicitement le passage à un autre thread pour garantir l'alternance :

```
public void run() {
  while (true) {
    System.out.println( Thread.currentThread().getName() );
    Thread.yield();
  }
}
```

Synchronisation

- Tout objet Java possède un verrou : un seul thread peut obtenir ce verrou à un instant donné.
- Une section critique est dite synchronisée sur un objet partagé lorsqu'elle utilise cette syntaxe :

```
synchronized(unObjet) { /* Section critique */ }
```

- Pour garantir qu'une méthode d'instance ne sera exécutée que par un seul thread à la fois (celui qui aura obtenu le verrou sur cette instance), il suffit de la déclarer avec le modificateur synchronized. Les deux définitions suivantes sont donc identiques:

```
synchronized methode(...) { /* Section Critique */ }
methode(...) { synchronized(this) { /* Section Critique */ } }
```

Threads et signaux

- La classe Object définit les trois méthodes wait(), notify() et notifyAll().
- Le thread qui appelle wait() sur un objet est ajouté à la liste des threads en attente sur cet objet : son exécution est interrompue.
- Un thread appellant notify() sur cet objet réveillera un des threads en attente sur cet objet (FIFO non garanti).
- Un appel à notifyAll() réveille tous les threads en attente sur l'objet (ils concourront pour obtenir le verrou).

Exemple : producteur/consommateur

```
public class Producteur extends Thread {
  private Tampon  _tampon;
  private int  _numero;

  public Producteur(Tampon tampon, int num) { _tampon = tampon;  _numero = num; }

  public void run() {
    for (int i = 0; i < 10; i++) {
      _tampon.put(i);
      System.out.println("Le producteur n° " + _numero + " a écrit " + i);
      try {
        sleep((int)(Math.random() * 100));
      } catch(InterruptedException e)  { }
    }
  }
}

public class Consommateur extends Thread {
  private Tampon  _tampon;
  private int  _numero;

  public Consommateur(Tampon tampon, int num) { _tampon = tampon;  _numero = num; }

  public void run() {
    int valeur;
    for (int i = 0; i < 10; i++) {
      valeur = _tampon.get();
      System.out.println("Le consommateur n° " + _numero + " a lu " + valeur);
    }
  }
}
```

Exemple : producteur/consommateur

```
public class Tampon {
  private int _contenu;
  private boolean _dispo = false;

  public synchronized int get() {
    while (!_dispo) {
      try { wait(); } catch(InterruptedException e) { }
    }
    _dispo = false;
    notifyAll();
    return _contenu;
  }
  public synchronized void put(int val) {
    while (_dispo) {
      try { wait(); } catch(InterruptedException e) { }
    }
    _contenu = val;
    _dispo = true;
    notifyAll();
  }
}

public class TestProdCons {
  public static void main(String[] args) {
    Tampon tampon = new Tampon();
    Producteur prod = new Producteur(Tampon, 1);
    Consommateur cons = new Consommateur(Tampon, 1);

    prod.start();
    cons.start();
   }
}
```

Entrées/Sorties

Octets et caractères

- Les classes d'entrées/Sorties classiques sont définies dans le paquetage java.io.
- java.io fait la différence entre flux d'octets (classes abstraites InputStream et OutputStream) et flux de caractères (classes abstraites Reader et Writer).
- On peut passer d'un flux d'octets à un flux de caractères (classes InputStreamReader et OutputStreamWriter).
- java.io définit également la classe File (pour manipuler un fichier physique) et la classe IOException et ses filles.

Entrées/sorties de caractères

- Les entrées/sorties standard d'un programme Java sont reliées aux objets statiques System.in, System.out et System.err (System est une classe de java.lang).
- System.in peut être assimilé à un InputStream, System.out et System.err à des OutputStream.
- Pour lire des caractères on utilise le plus souvent un BufferedReader or, pour construire un BufferedReader, il faut un Reader.

Lecture de caractères

- Pour construire un Reader à partir d'un Stream on peut utiliser InputStreamReader.
- Pour construire un Reader à partir d'un fichier texte, on peut utiliser un FileReader.

```
BufferedReader clavier, fichier;
try {
 clavier = new BufferedReader(new
                                    InputStreamReader(System.in));
 fichier = new BufferedReader(new FileReader("machin.txt"));
 .....
 clavier.close(); fichier.close();
} catch(IOException e) {.....}
```

Exemple de lecture sur System.in

```
import java.io.*;
/** Lecture d'un entier au clavier */
public class ReadIntSurStdin {
  public static void main (String[] args) {
    String ligne;
    int valeur;      // Initialisation nécessaire pour le println
    try {
      BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
      ligne = is.readLine();
      valeur = Integer.parseInt(ligne);    // ou : valeur = Integer.parseInt(is.readLine());
    } catch (NumberFormatException e) {
       System.err.println("Nombre incorrect : " + e);
    } catch (IOException e) {
       System.err.println("Erreur d'entrée/sortie: " + e);
    }
    System.out.println("Nombre lu : " + valeur);
  }
}
```

Entrées/Sorties d'octets

- On utilisera des FileInputStream ou des FileOutputStream pour les E/S sur fichiers.
- On utilisera des BufferedInputStream ou des BufferedOutputStream pour effectuer des E/S tamponnées.
- On utilisera des DataInputStream et des DataOutputStream pour des E/S de données binaires.

Exemple

```
import java.io.*;

public class CopieFichiers {
   public static void main(String[] args) {
      if (args.length != 2) {
                System.err.println("Usage: java CopieFichiers <source> <dest>");
         System.exit(1);
      }

      BufferedInputStream is;
      BufferedOutputStream os;

      try {
         is = new BufferedInputStream(new FileInputStream(args[0]));
         os = new BufferedOutputStream(new FileOutputStream(args[1]));
         int octet;

         while ((octet = is.read()) != -1)
              os.write(octet);

         is.close();
         os.close();
      } catch(FileNotFoundException e) {
         System.err.println(e);
      } catch(IOException e) {
         System.err.println(e);
      }
   }
}
```

Exemple

```
import java.io.*;

public class ReadWriteBin {
   // Ici main() propage les exceptions (voir différences avec exemple précédent)
   public static void main(String[] args) throws IOException {
     if (args.length != 1) {
                System.err.println("Usage: java ReadWriteBin <fichier>");
        System.exit(1);
      }

     DataOutputStream os = new DataOutputStream(new FileOutputStream(args[0]));

     int unEntier = 10;
     double unReel = Math.PI;
     os.writeInt(unEntier);
     os.writeDouble(unReel);
     os.close();

     /* Vérification */
     DataInputStream is = new DataInputStream(new FileInputStream(args[0]));

     unEntier = is.readInt();
     unReel = is.readDouble();
     is.close();

     System.out.println("unEntier = " + unEntier + ", unReel = " + unReel);
   }
}
```

Sérialisation

- La sérialisation d'objet est un moyen automatique de sauvegarde et de relecture de l'état d'un objet.
- Tout objet d'une classe implémentant l'interface Serializable peut être sauvegardé et restauré sur ou à partir d'un stream (ObjectOutputStream définit writeObject() et ObjectInputStream définit readObject()).
- La sérialisation par défaut sauvegarde la valeur des variables d'instance "non transient".

Sérialisation

- Quand un objet est sérialisé, toutes les références d'objets qu'il contient le sont également.
- Ceci implique qu'un objet sérialisé ne doit contenir que des références à des objets Serializable !
- On peut contrôler cela en marquant les membres non sérialisables comme transient, ou alors en redéfinissant le mécanisme de sérialisation.
- Une variable d'instance transient indique que son contenu est inutile en dehors du contexte courant et qu'il ne doit jamais être sauvegardé.

Exemple

```
import java.io.*;
import java.util.*;

/* Création d'une table de hachage et sauvegarde dans un fichier.
  Un Hashtable est sérialisable car la classe Hashtable implémente
  Serializable */

public class Sauve {
   public static void main(String[] args) throws IOException, ClassNotFoundException {

      Hashtable hash = new Hashtable();

      hash.put("chaine", "Louis Ferdinand Céline");
      hash.put("entier", new Integer(26));
      hash.put("rÈel", new Double(Math.PI));

      ObjectOutputStream out = new ObjectOutputStream(new
                                                      FileOutputStream("hash.save"));
      out.writeObject(hash);
      out.close();

      /* Verif */
      ObjectInputStream in = new ObjectInputStream(new
                                                      FileInputStream("hash.save"));
      Hashtable autreHash = (Hashtable)in.readObject();
      in.close();

      System.out.println("Hachage relu : " + autreHash);
   }
}
```

Autre exemple

```
import java.io.*;
import java.util.*;

class MaClasse implements Serializable {
   private String userName;
   private String motDePasseChiffré;
   private transient String motDePasseEnClair;

   public MaClasse(String user, String code) {
          userName = user;
      motDePasseEnClair = code;
     // motDePasseChiffré = DES.encrypt(code);
      motDePasseChiffré = code + " est chiffré !"; // on simule...
   }

   public String toString() {
          return userName + " (" + motDePasseChiffré + ")";
   }
}
```

Autre exemple (suite)

```
public class Serialise {
   public static void main(String[] args) throws IOException, ClassNotFoundException {
      ArrayList liste = new ArrayList();
      MaClasse unObjet = new MaClasse("toto", "titi");

      liste.add(new Date());
      liste.add(new MaClasse("jaco", "secret"));
      liste.add(unObjet);

      ObjectOutputStream out = new ObjectOutputStream(new
                                                FileOutputStream("users.dat"));
      out.writeObject(liste);
      out.close();

      /* Verif */
      ObjectInputStream in = new ObjectInputStream(new
                                                FileInputStream("users.dat"));
      ArrayList autreListe = (ArrayList)in.readObject();
      in.close();

      System.out.println("Liste relue : " + autreListe);
   }
}
```