Chapitre 8 : Java avancé

- Refactoring
- Gestion des dépendances
- Création et utilisation de bibliothèques
- Exceptions
- Duplication d’objets
- Entrées/Sorties
- Threads
- Javadoc

Refactoring

- **Déplacer**, **renommer** une classe
- **Renommer** un attribut, une méthode
- **Créer** un attribut, une variable à partir d'une valeur
- **Générer** une méthode à partir de code
- **modifier** la signature d'une méthode
- **Faire remonter** un attribut dans la classe mère
- **Faire descendre** un attribut dans une classe fille
- **Extraire** l'interface

Gestion des dépendances

- Gestion des packages
  - Un package rassemble :
    - des classes et/ou des interfaces
    - des sous-packages

  exemples de packages :
    - java.lang : classes de base String, Integer…
    - java.util : classes utilitaires Collections, Timer…

- En java :
  - Une classe (ou une interface) = un fichier
  - Un package = un répertoire

Gestion des dépendances

- Importer des classes et des packages
  - Dans un programme Java, toutes les classes définies dans le même package ou dans le package java.lang sont incluses implicitement.
  - Pour utiliser toute autre classe, il faut l'importer explicitement avec la directive **import**

Gestion des dépendances

- Importation d'une classe ou d'un package
  - `import monpackage.UneClasse;`
  - `import monpackage.*;`

  Note : l'instruction `import monpackage.*` n'importe pas les classes des sous-packages

- Indiquer le package d'une classe :

```
package packagedelaclasse;
```

Note : cette instruction est la première du fichier décrivant la classe ou l'interface

Création de bibliothèques

- L'archiver JAR (Java Archive) :
  - L'outil **jar** permet de construire des bibliothèques (des fichiers .jar) à partir de bytecode (et d'autres fichiers, par ex. des images).

- Création d'une archive à partir des fichiers du répertoire courant :

```
jar cvf monarchive.jar .
```

Utilisation de bibliothèques

- Compiler en utilisant un jar :

```
javac -classpath monjar.jar monprojet/*.java
```

- Lancer un programme qui utilise un jar :

```
java –classpath monjar.jar: monprojet/Main
```

Note : le classpath indique l'ensemble des répertoires (séparés par :) qui sont explorés pour compiler ou exécuter un programme java.

Le classpath

- Deux manières de définir le classpath :
  - Il peut être défini pour chaque appel d'un outil du JDK (javac, java, jdb…)avec l'option –classpath

    cf. exemple diapo précédente

  - Il peut être défini par la variable d'environnement CLASSPATH

    exemple bash

    ```
    export CLASSPATH=path1:path2…
    ```

Exceptions

- Qu'est ce que c'est ?
  - C'est un signal qui intervient au cours de l'exécution d'un programme et qui indique qu'une instruction ne s'est pas déroulée normalement.

- A quoi ça sert ?
  - gérer les erreurs qui peuvent intervenir dans le déroulement d'un programme.

Ne pas gérer les erreurs peut avoir de graves conséquences

Exceptions

- Quelques exceptions disponibles :
  - `CloneNotSupportedException`
    La classe d'un objet qui appelle la méthode clone() n'implémente pas l'interface Cloneable
  - `NullPointerException`
    Utilisation non autorisée de l'objet null. Par exemple :
    ```
    Voiture v;
    v.getCouleur();
    ```
  - `ArithmeticException`
    Par exemple la division d'un entier par 0 retourne une instance de cette exception

Exceptions

- Quelques exceptions disponibles :
  - `ClassCastException`
    Si on tente de caster un objet dans un type non compatible
  - `NegativeArraySizeException`
    Si on tente de créer un tableau avec une taille négative
  - `IndexOutOfBoundsException`
    Problème d'indice pour un tableau, un String ou une liste…

Exceptions

- Créer une nouvelle classe exception :

```
public class MonException extends Exception {
   …
}
```

- Exemple : PointException.java

Exceptions

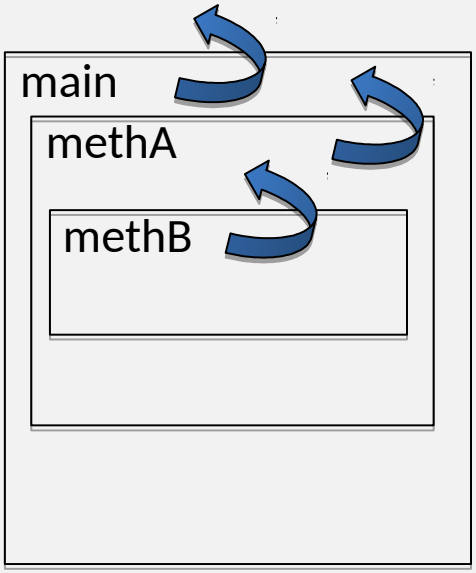
- Lancer une exception avec **throw :**

```
if (…) {
   throw new MonException();
}
```

Exceptions

- Que faire d’une exception ?
  - Deux choix possibles :
    - 1) la propager
    - 2) la traiter

Exceptions



- Propager une exception :
  - on indique que la méthode peut lancer des exceptions :

```
public void methodeA (…) throws MonException {
   …
 }
```

Note : une méthode peut lancer différents types d'exceptions. On indique alors les exceptions séparées par une virgule.

Exceptions

- Récupérer une exception :

Si une instruction peut lancer une exception de type MonException, alors on peut protéger son appel par un bloc **try catch** :

```
public void methodeB(…){
 …
 try {
        méthodeA(…);
  }
 catch (MonException e) {
      //ici, on a récupéré l'exception
 }
}
```

Exceptions

- Le mot clé **finally** (optionnel):
  - le bloc **finally** est exécuté à la fin du bloc try.
  - Cela permet de :
    - **factoriser du code** qui serait sinon dupliqué dans différents blocs catch
    - **effectuer des instructions après le bloc try** même si une exception a été lancée et non catchée

Exceptions

- Le mot clé **finally** :

```
public void methodeB(…){
 …
 try {
      //instructions;
  }
 catch (…) {
 //instructions;
 }
 catch (…) {
 //instructions;
 }
 …
 finally{
 // instructions;
 }
}
```

Dupliquer un objet

Tous les attributs sont dupliqués

- Copie de surface Shallow copy :

  Si des attributs sont des références, alors seule la référence est dupliquée.

- Copie en profondeur Deep copy :

  Si des attributs sont des références (mutables), alors les objets référencés sont dupliqués en deep copy également.

Dupliquer un objet

exemple : duplication de MyClass obj

- **Constructeurs par copie**

```
public MyClass(MyClass obj)
```

Deep copy : il faut faire appel aux constructeurs par copie des attributs (références mutables)

Dupliquer un objet

exemple : duplication de MyClass obj

- **Implémenter l'interface cloneable**

  redéfinir la méthode `clone` (de `Object`).

```
MyClass copy = obj.clone()
```

  Si Myclass n'implémente pas `Cloneable` : `CloneNotSupportedException`

  Deep copy : il faut aussi cloner les attributs

Dupliquer un objet

exemple : duplication de MyClass obj

- **Sérialization** (cf. la suite)
  - Permet de faire une deep copy.

Les Entrées/Sorties

- Les entrées/sorties sont basées sur la notion de flux (**stream**).

- Un flux est un ensemble de données (e.g. flux de caractères, de bytes, d'objets…) dirigé depuis ou vers le programme

- Manipulation de flux :
  - création
  - lecture
  - écriture
  - fermeture

Les Entrées/Sorties

- Le package **java.io** permet de manipuler les flux

- Deux types de flux :
  - flux entrants (**InputStream**) sont lus depuis une entrée (e.g. clavier, fichier…)
  - flux sortants (**OutputStream**) sont écrits vers une sortie (écran, fichier…)

Les Entrées/Sorties

- Il existe des flux déjà ouverts
  - System.in (`InputStream`): flux (byte) d'entrée standard
  - System.out (`PrintStream`) : flux (char) de sortie standard
  - System.err (`PrintStream`) : flux (char) d'erreur

- Exemple : lire un flux de caractères sur l'entrée standard et réécrire ce flux sur la sortie standard

```
int c;
c=System.in.read();
while(c!=-1){
  System.out.println((char)c);
  c=System.in.read();
}
```

Les Entrées/Sorties

- Utilisation de **Reader** (resp. **Writer**) pour lire (resp.écrire) dans les flux d'entrée (resp.de sortie) de caractères

```
int c=0;
Reader r = new InputStreamReader(System.in);
Writer w = new OutputStreamWriter(System.out);
c = r.read();
while(true){
  w.write(c);
  c=r.read();
  if(c==-1){
  break;
  }
}
w.flush();
```

Note : l'utilisation d'un `InputStreamReader` et d'un `OutputStreamWriter` permet de spécifier l'encodage de caractères utilisé (UTF-8, ISO-8859-1…)

Les Entrées/Sorties

- Exemple d'E/S sur un fichier

```
int c=0;
Reader r = new FileReader("toto_in.txt");
Writer w = new FileWriter("toto_out.txt");
c = r.read();
while(true){
  w.write(c);
  c=r.read();
  if(c==-1){break;}
}
w.close();
```

Les E/S optimisées

- Lectures/écritures optimisées (buffer) avec les **BufferedReader** et les **BufferedWriter**

```
Reader r = new BufferedReader(new
  FileReader("toto_in.txt"));
Writer r = new BufferedWriter(new
  FileWriter("toto_out.txt"));
```

- Attention : le caractère c n'est pas écrit immédiatement après l'appel de `write(c)` mais seulement lorsque le buffer est vidé.

Les E/S de types primitifs et de Strings

- La classe **Scanner** permet de parser des flux de types primitifs (int, char, boolean) mais aussi des flux de chaînes de caractères (String).

```
//exemple à partir de l'entrée standard
Scanner sc1 = new Scanner(System.in);

//exemple à partir d'une chaîne
Scanner sc2 = new Scanner («coucou ca va 123 ?»);

//exemple à partir d'un fichier
Scanner sc3 = new Scanner (new File("toto_in.txt"));
```

Les E/S de types primitifs et de Strings

- Quelques méthodes de la classe **Scanner**

  - `hasNext()` : retourne vrai s'il reste un lexème dans le flux
  - `hasNextInt()` : retourne vrai si le prochain lexème peut être interprété comme un int
  - `next()` : retourne le prochain lexème (par défaut, délimité par un caractère d'espacement (`Character.isWhitespace()`)
  - `nextInt()` : retourne l'entier correspondant au lexème suivant
  - `useDelemiter(String pattern)` : permet de changer le délimiteur des lexèmes par une expression régulière

Pour plus d'infos, ne pas hésiter à consulter la doc…

http://download.oracle.com/javase/7/docs/api/

Les E/S d'objets

- Création de flux d'objets
  - **ObjectInputStream** : flux d'entrée
  - **ObjectOutputStream** : flux de sortie

Sérialization : enregistrement d'un objet dans un flux

Désérialization : récupération d'un objet d'un flux

Pour pouvoir être sérialisé, un objet doit implémenter l'interface (vide) Serializable. Sinon, une exception sera levée.

Les E/S d'objets

- La **Sérialization :**
  - Création d'un flux de sortie d'objets :

```
ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("toto.ser"));
```

  - Sérialization des objets :

```
oos.writeObject(maVoiture);
oos.writeObject(unCercle);
```

Note : On peut sérialiser des objets de différents types dans un même flux. Mais lors de la désérialization, il faudra procéder dans le même ordre.

Les E/S d'objets

- La **Désérialization :**
  - Création d'un flux d'entrée d'objets :

```
ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("toto.ser"));
```

  - Désérialization des objets (dans le même ordre):

```
Voiture v = (Voiture)ois.readObject();
Cercle c = (Cercle)ois.readObject();
```

Note : La méthode readObject() retourne un `Object`

Les E/S d'objets

- La sérialization d'un objet entraine la sérialization de tous ses attributs à l'exception des transient. Ils doivent donc être également sérializables.

- transient : empêche la sérialization d'un attribut

- Sérialization : deep copy
  - *(cf. in-memory serialization)*

Les Threads - Généralités

- Un **thread** (lightweight process) est un fil d'instructions.

- Chaque programme JAVA possède au moins un thread principal **main**, dans lequel il est possible d'en créer des nouveaux.

- Chaque Thread possède sa propre pile d'exécution.

Les Threads - Généralités

- Les ordinateurs possédant des processeurs multi cœurs permettent d'exécuter **en même temps** différents threads.

- La JVM s'occupe d'**ordonnancer** les différents threads, c'est à dire de leur donner la main à tour de rôle.

Les Threads – Création

- Différents états d'un Thread
  - Nouveau (NEW) : Thread créé, mais pas encore lancé
  - En cours (RUNNABLE) : Thread en cours d'exécution
  - Bloqué (BLOCKED, WAITING, TIMED_WAITING) : Thread mis en attente
  - Terminé (TERMINATED) : Thread terminé, lorsque la méthode start() est achevée

Les Threads – Création

- Deux manières pour créer un thread :

  – Hériter de la classe Thread

```
public class MyThread extends Thread{
…
}
```

  – Implémenter l'interface Runnable et instancier un Thread via le constructeur `Thread(Runnable r)`

NOTE : Dans les deux cas, ce que fait le thread est décrit dans la méthode run()

Les Threads – Création

- Dériver la classe Thread

```
public class MyThread extends Thread {
    int cpt;

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            incCpt();
            System.out.println(Thread.currentThread().getName() + ": " + cpt);
        }
    }

    public void incCpt() {cpt++;}

    public static void main(String[] args) {
        Thread th1 = new MyThread();
        Thread th2 = new MyThread();
        th1.start();
        th2.start();
    }
}
```

Les Threads – Création

- Implémenter de l'interface Runnable

```
public class MyRunnable implements Runnable{
    int cpt;

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            incCpt();
            System.out.println(Thread.currentThread().getName() + ": " + cpt);
        }
    }

    public void incCpt() {cpt++;}

    public static void main(String[] args) {
        Runnable run = new MyRunnable();
        Thread th1 = new Thread(run);
        Thread th2 = new Thread(run);
        th1.start();
        th2.start();
    }
}
```

Les Threads – Création

- Avantages / Inconvénients

  – Si on dérive la classe Thread, on ne peut plus hériter d'une autre classe (pas d'héritage multiple en JAVA)

  – Au contraire, lorsque l'on implémente l'interface Runnable, on peut encore hériter d'une classe.

ATTENTION aux accès concurrents !

  – Il est possible de créer deux threads à partir du même objet Runnable

  – On peut également avoir des données communes entre les threads.

  exemple : MyRunnable.java

Les Threads – Accès concurrents

- Le mot clé **synchronized**
  - permet d'indiquer de que certaines instructions doivent être effectuées par un seul thread à la fois.

```
synchronized(unObjet){
   // ce bloc sera effectué
   entièrement par un thread
}
```

Les Threads – Accès concurrents

- Si le bloc synchronisé concerne une méthode complète, on peut écrire :

```
public synchronized void uneMethode(){
…
}
```

- QUESTION : que se passe t-il si deux threads manipulent un même objet avec une méthode synchronized et l'autre pas ?

Les Threads – méthodes utiles

- `static Thread currentThread()` : retourne une référence sur le Thread courant
- `static void sleep(long l)` : met en pause le thread durant l millisecondes
- `public void start()` : appel de la méthode run()
- `public State getState()` : retourne l'état du Thread
- `public String getName()` : retourne le nom du Thread
- `public boolean isAlive()` : retourne vrai si le thread a été lancé mais n'est pas encore terminé

Les Threads - wait() et notify()

- Nous venons de voir comment régler les problèmes d'accès concurrents sur des données partagées.

- Il peut également être utile de pouvoir faire communiquer les threads entre eux afin par exemple qu'un thread attende qu'un autre ait effectué une certaine tache avant de continuer

- On peut dire explicitement à un thread de se mettre en pause à l'aide de l'instruction **wait()**

- On peut envoyer le signal **notify()** afin de réveiller un thread qui est en pause. Le signal **notifyAll()** réveille tous les threads en pause.

Les Threads - wait() et notify()

- **Remarques :**
  - On n'a aucun contrôle sur le thread qui sera réveillé par l'appel de **notify()**.
  - Si un signal **notify()** est lancé alors qu'aucun thread n'est en pause, alors ce signal est perdu. Cela peut provoquer des situations de bloquage où un thread attend en permanence un **notify()** qui n'arrivera jamais…
  - Les méthodes **wait()** et **notify()** s'exécutent dans des méthodes synchronisées

Javadoc

- Outil permettant de générer de la documentation html à partir du code source.

- Principe :
  - Un format spécial de commentaires sont interprétés pour générer la javadoc.

```
/**
* Commentaires utilisés
* pour générer la javadoc
*/
```

Javadoc

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

- Quelques tags utilisés par Javadoc :
  - `@author : indique l'auteur d'une classe.`
  - `@version : indique la version de la classe.`
  - `@see : permet de faire référence à une section de la javadoc (un attribut, une méthode, un constructeur, une classe ou package)`

Javadoc

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

- Quelques tags utilisés par Javadoc :
  - `@throws : indique le type d'exception que retourne la méthode`
  - `@param : permet de décrire les paramètres de la méthode`
  - `@return : permet de décrire la valeur de retour de la fonction`