Tutoriel log4j : A propos de le journalisation …
(traces ou messages ou logs)

Ce tutoriel n'est à effectuer qu'après la mise en place dans le projet `cabinetMedical` de la sérialisation/désérialisation dans un fichier d'une liste de `Personne`, c'est-à-dire une fois que la classe `PersonneDAOFichier` est correctement écrite et testée !

Dans la phase de développement, pour observer le comportement d'un programme, les développeurs placent souvent ***des traces*** dans le code en utilisant la sortie standard (`out`) ou la sortie erreur (`err`) et la classe statique `System`, comme le montre le code suivant :

```
System.out.println("Une trace dans la console Java");
System.err.println("Une autre trace en rouge dans la console Java");
```

Ce genre d'instructions, bien que permettant de trouver des erreurs, complique le cycle de développement : par exemple il faut penser à effacer ou à commenter ces instructions quand le programme est fini. C'est pourquoi les bonnes pratiques de développement actuelles déconseillent l'utilisation des méthodes `System.out.print*` et `System.err.print*` pour afficher des messages et recommandent plutôt ***l'utilisation d'un outil de journalisation*** (*outil* au sens de framework, API, …)
Un ***log*** (ou ***journal***) contient des messages qui permettent d'avoir un suivi a postériori sur le "bon" déroulement d'une application.Typiquement chaque message de ***log*** comporte : un niveau de gravité (ou criticité), la date (avec l'heure à laquelle il a été émis), un message à personnaliser (comprenant la nature de l'événement au travers d'une description, et éventuellement d'autres informations : utilisateur, classe, etc…). La ***journalisation*** consiste donc à garder les traces sur un support sûr des évènements survenus dans un système ou dans une application.
Un ***outil de journalisation*** va permettre entre autres, grâce à son(ses) fichier(s) de configuration, d'activer ou de désactiver à tout moment certains messages de traces en fonction des besoins sans avoir besoin d'intervenir dans le code déjà écrit. En effet, la conception de projet nécessite la ***mise en place de traces*** lors des phases de ***développement*** (notamment pour le déboggage)… mais aussi lors de la mise en ***production***.

Les outils de journalisation vont offrir de nombreux avantages aux développeurs.
Le service le plus utilisé est la mise au point du code lors de la phase de développement (c'est ce que nous verrons dans ce tutoriel). Mais ils permettent aussi d'enregistrer des messages, d'envoyer des emails, de gérer des niveaux de traces, mais surtout par l'intermédiaire d'un fichier de configuration de gérer à tout moment les traces.

Installation de l'API `log4j` :

L'API **`log4j`** est l'API de journalisation généralement utilisée en entreprise.
`log4j` est développée par la fondation Apache.
Pour ce TP, l'archive **`jar`** de la bibliothèque **`Log4J`** est disponible sur la zone libre ou depuis le site : http://logging.apache.org/log4j/docs/download.html
En téléchargeant la dernière version stable et en la dézippant vous trouverez entre autres le fichier **`log4j-x.x.x.jar`** où **`x.x.x`** est le numéro de version)

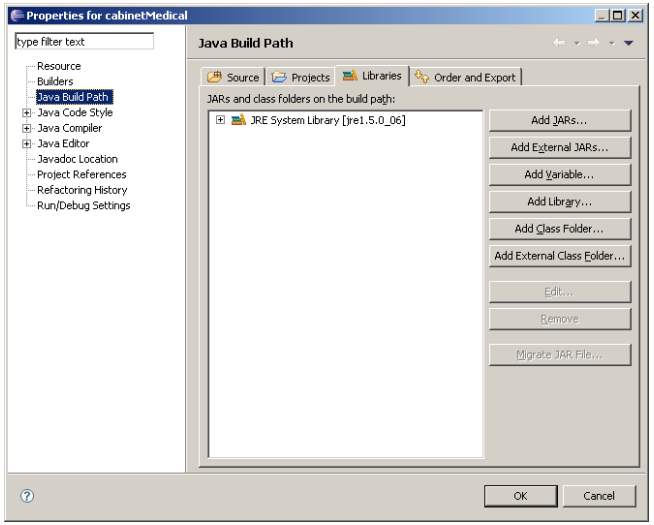
→ Pour commencer, nous devons ajouter à notre projet un nouveau dossier **`lib`** qui contiendra l'archive **`jar`** de **`log4j`**.
Pour cela, placez-vous dans le vue **`Package`** . Une fois le projet **`cabinetMedical`** sélectionné, effectuez un **clic droit** puis choisir **`New…`** puis **`Folder`**.
Dans **`Folder Name`** , tapez **`lib`** et validez.

Placez-vous alors sur le nouveau répertoire **`lib`** créé, **clic droit** et sélectionner **`Import`**… pour importer dans ce répertoire, le fichier **`jar`** (**`log4j-x.x.x.jar`**) disponible sur la zone libre.

→ Pour pouvoir utiliser **`log4j`** dans notre projet, il est nécessaire d'inclure le fichier **`log4j-x.x.x.jar`** dans le **`classpath`.**

Pour cela, replacez-vous dans la vue **`Package`** sur le projet **`cabinetMedical`** :

Effectuer un clic droit avec la souris et choisir l'option **`Properties`**
La fenêtre ci-contre doit s'ouvrir.



Sélectionnez à gauche **`Java Build Path`**

Puis, cliquez à droite sur le bouton **`AddJARs…`**
(ou **`Add External JARs…`**)

Sélectionnez le fichier **`log4j-x.x.x.jar`** que vous venez d'importer, c'est-à-dire celui disponible dans le répertoire **`lib`** du projet **`cabinetMedical`**.

Validez et cliquez sur **`OK`**.

Présentation de l'API `log4j` :

`log4j` met 3 composants à disposition du développeur :
→ les composants de type **`Logger`** qui permettent d'écrire les messages
→ les composants de type **`Appender`** qui permettent de sélectionner la destination des messages
→ les composants de type **`Layout`** qui permettent de mettre en forme les messages

L'entité de base : le `Logger` correspondant au *journal*

Le *Logger* est l'entité de base pour effectuer la journalisation, il est mis en oeuvre par le biais de la classe **`org.apache.log4J.Logger`**
L'obtention d'une instance de type `Logger` se fait en appelant la méthode statique **`getLogger`** :

```
Logger logger = Logger.getLogger("com.iut.cabinet.metier.PersonneDAOFichier");
```

La déclaration d'un **`Logger`** sera réalisé dans chaque classe qui utilisera le système de journalisation.
On peut donner n'importe quel nom au **`Logger`** mais il est recommandé de donner au `Logger` le nom complet de la classe où le *logger* a été créé. Ainsi, chaque classe aura son journal.
On peut aussi donner le nom du package pour que toutes les classes utilisent le même journal : c'est une configuration un peu moins fine.

Messages à journaliser

Afin de donner une importance au message dans le journal, chaque message sera enregistré avec ***un niveau de journalisation*** appelé aussi ***priorité des messages.***
La classe **`org.apache.log4J.Level`** permet de gérer ces niveaux de journalisation en proposant différents niveaux de criticité (gravité) qui ont chacun un poids différent.
Le liste ci-dessous présente les niveaux du plus fort au plus faible.

- **`FATAL`** : Niveau utilisé pour une erreur grave pouvant provoquer **l'arrêt prématuré de l'application**
- **`ERROR`** : Niveau utilisé pour une **erreur** qui n'arrête pas prématurément l'application (requête SQL, copie de fichier, …)
- **`WARN`** : Niveau utilisé pour un **avertissement**
- **`INFO`** : Niveau utilisé pour des messages à caractère **informatif**
- **`DEBUG`** : Niveau utilisé pour générer des messages en phase de **déboggage**
- **`TRACE`** : Niveau utilisé pour générer des messages **de traces** d'exécution (disponible uniquement depuis la version 1.2.12)

Deux autres niveaux particuliers peuvent être utilisés à des fins de configuration.

- **`OFF`** : Aucun niveau de criticité n'est pris en compte
- **`ALL`** : Tous les niveaux sont pris en compte c-a-d ne filtre aucun message

Si vous aviez besoin de niveaux supplémentaires, vous pourriez envisager de créer les vôtres en sous-classant la classe **`org.apache.log4j.Level`**, néanmoins ceux déjà proposés dans la classe **`Level`** devraient être suffisants.

Remarque :

- On peut d'ores et déjà noter qu'un message n'est journalisé que si son niveau de journalisation est supérieur ou égal à celui du `Logger` effectuant la journalisation.
  Par exemple, durant la période de développement, on utilise le niveau DEBUG pour le `Logger`.
  Et une fois l'application déployée en production, il suffit juste de changer le niveau en WARN afin d'**ignorer** les messages de debugage, **sans modifier aucune autre ligne dans notre code !** (on y reviendra un peu plus loin dans le tutoriel)
- Les versions antérieures à la version 1.2 de `log4J` utilisaient la classe `Category` pour gérer les messages et la classe `Priority` pour encapsuler les niveaux de gravité. Depuis la version 1.2, les classes `Category` et `Priority` sont remplacées respectivement par les classes **`Logger`** et **`Level`** sur lesquelles nous travaillerons.

→ ***Pour poster un message dans le journal***, il faut utiliser la méthode :

```
log(Priority level,String message)
```

Le premier argument ***`level`*** définit le niveau de criticité du ***`message`*** (second argument).

Exemple: Si l'on souhaite pour une phase de déboggage tracer l'entrée dans la méthode `storeAllPersonne,` on peut utiliser l'instruction suivante :

```
logger.log(Level.DEBUG,"storeAllPersonnes : Entree");
```

→ ***Pour simplifier l'envoi de messages au journal***, des ***alias*** de la fonction `log` ont été créés : il n'est pas nécessaire de spécifier le niveau de criticité car les alias portent pour identificateur le nom de ces niveaux.

```
void fatal(String msg), void error(String msg),
void warn(String msg), void info(String msg), void debug(String msg)
```

```
logger.debug("storeAllPersonnes : Entree");
```

signifie que l'appel à la méthode `debug` est équivalent à l'appel à la méthode `log` suivante :

```
logger.log(Level.DEBUG,"storeAllPersonnes : Entree");
```


→ Pour commencer, nous allons utiliser le `Logger` pour **tracer l'exécution du programme** :
Pour cela, nous enverrons des messages de niveau **`DEBUG`** au `Logger` au ***début et à la fin de chaque méthode***.

Travail à faire : Dans la classe **`PersonneDAOFichier`** :

→ Déclarer en **privé** le `Logger` suivant comme attribut de la classe `PersonneDAOFichier` :

```
Logger logger =
        Logger.getLogger(PersonneDAOFichier.class.getName());
```

Remarque : `System.out.println(`**`PersonneDAOFichier.class.getName()`**`);`
donnerait comme affichage à l'exécution : **`com.iut.cabinet.metier.PersonneDAOFichier`**

→ Vérifier que vous avez importé le bon package pour le journalisation c-a-d `log4j` dont l'**`import`** est :

```
import org.apache.log4j.Logger;
```

→ Pour commencer, nous nous intéresserons à la méthode permettant de réaliser la sérialisation c-a-d la méthode `storeAllPersonnes` . Complétez votre méthode afin que les premières et les dernières instructions de cette méthode permettent de mettre en place la journalisation grâce au `logger` :

```
public static void storeAllPersonnes (Collection<Personne> uneListe)
  {
  // Journalisation pour marquer l'entrée dans la méthode
  logger.debug ("storeAllPersonnes : Entree");


  ////////////////////////////////////////////
  // ... Votre code reste ensuite inchangé ...
  ////////////////////////////////////////////


  // Journalisation pour marquer la sortie de la méthode
  logger.debug("storeAllPersonnes : Sortie");

  } // fin storeAllPersonnes
```

Remarque : *N'oubliez pas de déclarer `logger` en **`static`**. Ceci est bien sûr propre à cette classe car le `logger` est utilisé dans une méthode statique `storeAllPersonnes` (…et comme vous le savez…les méthodes statiques ne peuvent utiliser que des variables statiques…)*

→ Exécuter pour tester votre code

A l'exécution, vous obtenez les messages d'erreur suivant :

```
log4j:WARN No appenders could be found for logger (com.iut.cabinet.metier.PersonneDAOFichier2).
log4j:WARN Please initialize the log4j system properly.
```

En effet, pour pouvoir journaliser avec `log4j`, il est nécessaire de configurer au préalable un **`Appender`** qui représente la cible d'un message, c-a-d l'endroit où le message sera stocké ou affiché.

L'Appender : flux de sortie pour un journal

Un **Appender** est une sortie utilisée pour enregistrer les événements de journalisation.
Chaque **Appender** a une façon spécifique d'enregistrer les événements.
Techniquement, **org.apache.log4j.Appender** est une **interface.**
Les classes qui définissent les **Appender** implémentent toutes l'interface **Appender**
log4j propose plusieurs types d'**Appender**, nous pouvons citer entre autres :

- org.apache.log4j.**ConsoleAppender** pour rediriger les messages du Logger vers la **console**
- org.apache.log4j.**FileAppender** pour rediriger les messages du Logger dans un **fichier**
- org.apache.log4j.**SMTPFileAppender** pour envoyer les messages du Logger par **mail**
- org.apache.log4j.**JDBCFileAppender** pour rediriger les messages du Logger vers **une base de données**
- etc …voir javadoc : http://logging.apache.org/log4j/1.2/apidocs/

Travail à faire : Afin de tracer le message d'erreur dans la console, associer en début de méthode **storeAllPersonnes** un **ConsoleAppender** au **logger** de la classe à l'aide de la méthode ***addAppender.***

```
public static void storeAllPersonnes (Collection<Personne> uneListe)
  {
  // Choix de l'Appender : sortie
  // Association du ConsoleAppender au logger de la classe
  ConsoleAppender stdout = new ConsoleAppender();
  logger.addAppender(stdout);

  // Journalisation pour marquer l'entrée dans la méthode
  logger.debug ("storeAllPersonnes : Entree");


  ////////////////////////////////////////////
  // ... Votre code reste ensuite inchangé ...
  ////////////////////////////////////////////


  // Journalisation pour marquer la sortie de la méthode
  logger.debug("storeAllPersonnes : Sortie");

} // fin storeAllPersonnes
```

→ Exécuter pour tester votre code

A l'exécution, vous obtenez encore un message d'erreur :

```
log4j:ERROR No output stream or file set for the appender named [null].
```

Pour pouvoir effectuer une journalisation, il est également indispensable de préciser la "présentation" utilisée par log4j pour l'écriture des messages. Pour cela, il est nécessaire de configurer un **Layout**.

Le Layout : mise en forme des événements de journalisation pour un journal

Les **Layout** sont utilisés avec les **Appender** afin d'associer la manière de tracer les données à la cible d'enregistrement.
Log4j fournit plusieurs **Layout**, nous pouvons citer:

- org.apache.log4j.**SimpleLayout** qui permet de journaliser de ***façon simple*** les événements. Le format d'affichage est **Niveau - Message**
- org.apache.log4j.**PatternLayout** qui permet de journaliser les messages en fonction d'un pattern (modèle) de données. Le pattern est composé de texte et de séquence d'échappement indiquant les informations à afficher (voir javadoc)
- org.apache.log4j.**HTMLLayout** qui permet de journaliser les messages au format HTML.
- org.apache.log4j.**XMLLayout** qui permet de journaliser les messages au format XML en conjugaison avec un **FileAppender.**

Travail à faire : Ajouter un **SimpleLayout** en paramètre au constructeur du **ConsoleAppender**.

```
public static void storeAllPersonnes (Collection<Personne> uneListe)
  {
  // Choix du Layout : format d'affichage
  SimpleLayout layout = new SimpleLayout();
  // Choix de l'Appender : sortie
  // Association du ConsoleAppender au logger de la classe
  ConsoleAppender stdout = new ConsoleAppender(layout);
  logger.addAppender(stdout);

  // Journalisation pour marquer l'entrée dans la méthode
  logger.debug ("storeAllPersonnes : Entree");


  ////////////////////////////////////////////
  // ... Votre code reste ensuite inchangé ...
  ////////////////////////////////////////////


  // Journalisation pour marquer la sortie de la méthode
  logger.debug("storeAllPersonnes : Sortie");

} // fin storeAllPersonnes
```

→ Exécuter pour tester votre code

A l'exécution, les deux messages de journalisation s'affichent enfin dans la console :

```
------------------------------------------------
DEBUG - storeAllPersonnes : Entree
DEBUG - storeAllPersonnes : Sortie
```

Remarque : Vous pouvez essayer de changer de Layout et de remplacer l'instruction :

```
SimpleLayout layout = new SimpleLayout();
```

par

```
PatternLayout layout = new PatternLayout("%d %-5p %c - %F:%L - %m%n");
```

Dans la console, ce format d'affichage des messages se traduit par un affichage de la date et l'heure, du niveau de journalisation, du nom du fichier et du numéro de ligne, et enfin du message personnalisé, suivi d'un retour à la ligne.

```
2009-10-20 16:00:07,984 DEBUG com.iut.cabinet.metier.PersonneDAOFichier - PersonneDAOFichier.java:36 - storeAllPersonnes : Entree
2009-10-20 16:00:08,015 DEBUG com.iut.cabinet.metier.PersonneDAOFichier - PersonneDAOFichier.java:55 - storeAllPersonnes : Sortie
```

Remarque : la classe **`PatternLayout`** permet ainsi de choisir précisément la mise en forme des messages de sortie afin d'informer au mieux les développeurs sur des données utiles…En contrepartie, elle demande parfois d'importantes ressources en fonction du détail des informations tracées.

Configuration dynamique de `log4j`

Avec `log4j`, il existe 3 méthodes pour configurer les loggers :
- méthode n°1 : ***configuration manuelle*** avec gestion des ressources directement dans le fichier source
- méthode n°2 : ***configuration dynamique*** avec gestion des ressources dans un **fichier de propriétés** au format habituel `cle=valeur`
- méthode n°3 : ***configuration dynamique*** avec gestion des ressources dans un ***fichier au format XML***

➢ ***1. La configuration manuelle de `log4j`*** est celle que nous venons de réaliser en paramétrant directement les `Appender` et les `Layout` avec des instructions java dans le fichier source… Mais elle n'est pas très pratique car elle mélange le code avec les éléments de configuration de la journalisation.

Configuration manuelle de `log4j`

```
// Choix du Layout : format d'affichage
SimpleLayout layout = new SimpleLayout();

// Choix de l'Appender : sortie
// Association du ConsoleAppender au logger de la classe
ConsoleAppender stdout = new ConsoleAppender(layout);
logger.addAppender(stdout);
```

➢ ***2 et 3. La configuration dynamique de `log4j`*** va être plus simple à manipuler. Il suffit dans un fichier de propriétés externes de configuer le `Logger` et de paramétrer les `Appender` et les `Layout`.
La configuration peut se faire par un ***fichier de propriétés classique*** (fichier *properties*) ou un ***fichier XML***.
La configuration par un fichier de propriétés est la première à avoir été implémentée : sa structuration est basique et ses possibilités sont plus restreintes que celles des fichiers XML. Nous ne l'aborderons pas dans ce tutoriel.
Pour la suite, nous adopterons ***la configuration dynamique par un fichier XML*** qui offre plus de possibilités et qui, de par son format, est plus structuré.

➢ ***3. La configuration dynamique de `log4j` par un fichier XML:***

Travail à faire : Dans votre projet `cabinetMedical`, depuis la vue Package d'Eclipse, clic droit et choisir **`New…Source Folder`** (passer par un Source Folder permet d'ajouter ce répertoire au classpath de votre application), créer alors un **répertoire `conf`** dans lequel vous importerez le fichier **`log4j.xml`** disponible sur la zone libre. Vérifier que le répertoire **`conf`** est créé au même niveau que le répertoire **`src`.**

→ Ouvrir ce fichier, nous allons le commenter …

Les ***fichiers XML*** sont beaucoup ***plus structurés*** que les fichiers de propriétés.
En effet, ils obligent à une ***certaine disposition*** des éléments puisque le fichier de configuration XML est validé vis-à-vis de sa ***DTD*** lorsqu'il est changé.

La syntaxe globale du fichier de configuration `log4j` au ***format XML*** est la suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <!--Déclaration des differents Appender et Loggers -->
</log4j:configuration>
```

La structure simplifiée d'un fichier de configuration `log4j` au ***format XML*** est la suivante :
→ configuration des `Appender` **`<appender> … </appender>`**
→ configuration des `Logger` **`<logger>…</logger>`**
→ configuration du `Logger` racine **`<root>…</root>`**

En effet, quelle que soit la configuration choisie (manuelle ou automatique), le procédé est toujours le même. Il faut tout d'abord configurer les `Appender`, puis vient la configuration des `Logger` (dont fait partie le Logger racine `root`)
Le minimum requis pour que `log4j` fonctionne correctement est d'attribuer un `Appender` correctement configuré au `Logger` racine `root`.

Nous allons d'abord décrire le contenu des balises XML **`<appender>`, `<logger>`** et **`<root>`,** puis nous illustrerons le fonctionnement de la configuration dynamique de **`log4j`** sur un exemple…

➢ Description de la configuration des `Appender` : balise `<appender>`

La configuration des **`Appender`** se fait au moyen de la balise **`appender`** qui peut prendre la forme suivante :

```
<appender name="NomAppender" class="ClasseImplementation">
 [<param name="ParametreAppender1" value="ValeurParametreAppender1"/>]
 [<layout class="ClasseLayout"<
     [<param name="ParametreLayout1" value="ValeurParametreAppender1"/>]
</layout>]
</appender>
```

L'attribut **`name`** permet d'indiquer un nom à l'`appender`. Ce nom sera utilisé dans la configuration des `Logger` pour faire référence à un `appender` défini.
L'attribut **`class`** permet d'indiquer la classe Java concrète qui implémente l'`appender`.

La balise **`layout`** est facultative pour l'**`appender`**.
La balise **`param`** est facultative aussi bien pour l'**`appender`** que pour le **`layout`**.

Dans le fichier `log4j.xml` proposé sur la zone libre, vous pouvez identifier **un `Appender`** : **`FileAppender`**, ce qui signifie que nous souhaitons pour notre projet que les messages de journalisation puissent être redirigés vers un **fichier** appelé ici **`cabinet.log`.**

→ **configuration de l'appender : `FileAppender` :**

sortie dans un fichier `cabinet.log` qui se trouvera dans un répertoire `log` avec un format personnalisé `PatternLayout`

`Log4j.xml`

```
<appender name="fichier" class="org.apache.log4j.FileAppender">
    <param name="file" value="./log/cabinet.log"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p] %C %M
[%L]: %m%n"/>
    </layout>
  </appender>
```

Remarque : Le chemin vers le fichier de `log` est ici défini de manière relative. La racine du chemin est alors le répertoire d'exécution .

➢ **Description de la configuration des `Logger` : balise `<logger>`**

La configuration des **`Logger`** se fait au moyen de la balise **`logger`** qui peut prendre la forme suivante :

```
<logger name="nomLogger" [additivity="false"]>
     [<level value="NomLevel"/>]
     [<appender-ref ref="NomAppender1"/>]
 </logger>
```

L'attribut **`name`** permet d'indiquer le nom du `Logger`, celui utilisé lors de l'appel à `getLogger`.
L'attribut **`additivity`** indique si l'additivité des `Appenders` s'applique ou non c-a-d si un logger d'un niveau donné va bénéficier de tous les Appenders de ces ancêtres en plus de ceux qui lui sont éventuellement affectés. Par défaut, l'additivité est active.

La balise **`level`** permet d'indiquer le niveau minimum qu'un message doit avoir pour être pris en compte par le `Logger`. Elle est facultative, si elle n'est pas précisé, le niveau est hérité du parent .
La balise **`appender-ref`** permet d'indiquer dans l'attribut **`ref`** le nom d'un `Appender` déclaré précédemment. Il peut y avoir plusieurs balises **`appender-ref`** pour un `logger`.

Dans le fichier `log4j.xml` proposé, vous ne trouvez pas pour l'instant de balise **`logger`**, nous en rajouterons une un peu plus tard dans le tutoriel (page 15), par contre, vous trouvez la balise **`root`** du logger racine
En effet, rappelons que le minimum requis pour que `log4j` fonctionne correctement est d'attribuer un `Appender` correctement configuré au `Logger` racine `root`.

➢ **Description de la configuration du `Logger` racine : balise `<root>`**

Le Logger racine est un `Logger` qui doit être configuré d'une façon particulière : Il n'a pas de nom attribué ni d'additivité (puisque c'est le log parent le plus haut) et est défini dans sa **propre balise `root` qui** reprend les balises `level` et `appender-ref` d'une balise `logger` classique.

Dans le fichier `log4j.xml` proposé, vous pouvez identifier **le *`logger` racine*** dans la balise ***`root`*** :
Ce `logger` accepte tous les logs de niveau minimum `FATAL` et les retransmet vers l'`Appender` de nom `fichier` défini précédemment (c-a-d le FileAppender associé au fichier `cabinet.log`).

`Log4j.xml`

```
<root>
    <level value="FATAL"/>
    <appender-ref ref="fichier"/>
  </root>
```

Si vous n'affectez pas de niveau de journalisation au *Logger* racine, il prend automatiquement la valeur **DEBUG**, cela peut conduire à l'affichage de nombreux messages, aussi, il est préférable de définir un niveau manuellement.

➢ ***Manipulation de la configuration dynamique de `log4j` : `log4j` par l'exemple ou comment paramétrer le fichier `log4j.xml` …***

L'`appender` défini dans le fichier de configuration indique que le fichier `cabinet.log` contenant les messages de journalisation (c-a-d les logs) sera stocké dans le répertoire **`log`**.

```
<param name="file" value="./log/cabinet.log"/>
```

Travail à faire : Dans votre projet `cabinetMedical`, depuis la vue Package d'Eclipse, clic droit et choisir **`New…Source Folder`**, créer alors un **répertoire `log`**.

Travail à faire : Pour tester, la configuration automatique, commencer par commenter (`/*...*/` et `//`) tout le code concernant les loggers déjà écrit dans la méthode `storeAllPersonnes`

```
    public static void storeAllPersonnes (Collection<Personne> uneListe)
      {
      /* // Choix du Layout : format d'affichage
      SimpleLayout layout = new SimpleLayout();
      // Choix de l'Appender : sortie -
      // Association du ConsoleAppender au logger de la classe
      ConsoleAppender stdout = new ConsoleAppender(layout);
      logger.addAppender(stdout);

      // Journalisation pour marquer l'entrée dans la méthode
      logger.debug ("storeAllPersonnes : Entree"); */

      ////////////////////////////////////////////
      // ... Votre code reste ensuite inchangé ...
      ////////////////////////////////////////////


      // Journalisation pour marquer la sortie de la méthode
      // logger.debug("storeAllPersonnes : Sortie");

} // fin storeAllPersonnes
```

➢ *Réglage du niveau de journalisation autorisé dans le logger racine `root`*

Travail à faire : Pour tester et manipuler la configuration automatique via des fichiers XML, nous allons implémenter le code suivant . Ce code est disponible sur la zone libre dans le fichier `TestLog4J.txt`

```
    public static void storeAllPersonnes (Collection<Personne> uneListe)
      {
      logger.trace("Test d'un message de log de niveau TRACE");
logger.debug("Test d'un message de log de niveau DEBUG");
      logger.info("Test d'un message de log de niveau INFO");
      logger.warn("Test d'un message de log de niveau WARN");
      logger.error("Test d'un message de log de niveau ERROR");
      logger.fatal("Test d'un message de log de niveau FATAL");

      ////////////////////////////////////////////
      // ... Votre code reste ensuite inchangé ...
      ////////////////////////////////////////////

} // fin storeAllPersonnes
```

→ Exécuter pour tester votre code

Ouvrir le fichier **`cabinet.log`** du répertoire **`log.`**

Que constatez-vous ?

En effet, seul le message de log de niveau **FATAL** a été enregistré, puisque le fichier **`log4j.xml`** a été paramétré de telle sorte que le logger **`root`** n'accepte que les messages dont le niveau mininum est **FATAL** :

```
<level value="FATAL"/>
```


Travail à faire : Modifier le fichier **`log4j.xml`** en effaçant la ligne `<level value="FATAL"/>` de l'élément **`root`**.

```
<root>
    <level value="FATAL"/>
    <appender-ref ref="fichier"/>
  </root>
```

→ Enregistrer le fichier **`log4j.xml`** exécuter le code et consulter le fichier **`cabinet.log`**

Ouvrir le fichier **`cabinet.log`**. ***Que constatez-vous ?***

Cette fois-ci tous les messages ont été loggués (hormis TRACE), puisque si aucun niveau de journalisation n'est affecté au *logger* **`root`**, il prend automatiquement la valeur **DEBUG** qui est le niveau le plus bas après TRACE.

Travail à faire : Modifier encore une fois le fichier **`log4j.xml`** en rajoutant un niveau minimum de **`WARN`** pour logguer les messages dans **`root`**.

```
<root>
    <level value="WARN"/>
    <appender-ref ref="fichier"/>
  </root>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Consulter le fichier **`cabinet.log`** et constater que seuls les messages de niveau égal ou supérieur à **`WARN`** sont loggués.

La balise **`<level>`** de l'élément **`root`** permet donc bien de définir un niveau minimal (filtre) pour tous les *logger*s c-a-d que tous les messages en dessous de ce niveau seront ignorés **quel que soit leur niveau** (mais les messages au-dessus de ce niveau sont traités normalement, en fonction du niveau de chaque *Logger*, voir plus loin).

➢ *Ajout d'un appender dans le fichier de configuration `log4j.xml`:*

Si vous souhaitez, visualiser les messages de log dans ***un fichier et dans la console***, rien de plus simple, il vous suffit d'ajouter un nouvel `appender` définissant la console, c-a-d un `ConsoleAppender`.

Travail à faire :

→ **Pour paramétrer un nouvel `appender`**, il suffit de rajouter une nouvelle balise **`appender`** juste après la balise `appender` définissant le `FileAppender` juste avant le logger `root`, c-a-d que vous devez rajouter le code ci-dessous (disponible dans **`TestLog4j.txt`**) entre les balises **`</appender>`** et **`<root>`** du fichier **`log4j.xml:`**

```
<appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.SimpleLayout">
    </layout>
 </appender>
```

→ **Pour associer l'`appender console` au logger `root`**, il suffit de rajouter dans la balise `root`,
au dessous : `<appender-ref ref="fichier"/>`
la ligne suivante : `<appender-ref ref="console"/>`

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.

Que constatez-vous ?
Les messages de log de niveau minimum **WARN** sont désormais à la fois mémorisés dans le **fichier** ***cabinet.log*** et à la fois affichés dans la **console**.

➢ Réglage du niveau de journalisation autorisé :

1. **Réglage au niveau du logger : balise `<level>`**

Le niveau de journalisation de **log4j** peut être défini de manière globale pour tous les appenders associés à un logger. C'est le rôle de la balise **<level>** qui est incluse dans la balise **<root>** (et qui pourra également être présente dans les balises de type **<logger>**)

2. **Réglage au niveau de l'`appender` : avec `threshold`**

Mais le niveau de journalisation peut aussi être redéfini au niveau de chaque **appender**, avec le paramètre de nom '**threshold**' dans une balise *<param>*

Travail à faire :
→ **Réglage au niveau WARN de l'`appender FileAppender`:**
Rajouter dans la balise **appender** du `FileAppender` une balise `<param>` indiquant que désormais le seuil des messages admis (`threshold`) est **WARN**:

```
<appender name="fichier" class="org.apache.log4j.FileAppender">
   <param name="file" value="./log/cabinet.log"/>
   <param name="threshold" value="warn"/>
    <layout class="org.apache.log4j.PatternLayout">
 <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p] %C %M [%L]: %m%n"/>
    </layout>
  </appender>
```

→ **Modifier le réglage du logger root en le repassant à DEBUG :**

```
<root>
   <level value="DEBUG"/>
    <appender-ref ref="fichier"/>
    <appender-ref ref="console"/>
  </root>
```

→ Enregistrer le fichier **log4j.xml** et exécuter le code.
Vérifier que tous les messages de log (hormis ceux de niveau TRACE) sont affichés dans la console et que seuls les messages de niveau égal ou supérieur à **WARN** sont affichés dans le fichier `cabinet.log`

➢ Mise en place d'une personnalisation de `logger` :

Dans le fichier de configuration de la journalisation (**log4j.xml**), nous ne manipulons jusqu'à présent qu'un seul logger, le logger racine (objet **RootLogger**) représenté dans le fichier **xml** par l'élément **root**. Tous les messages de journalisation issus de n'importe quelle classe de l'application tombent alors obligatoirement dans ce logger (**root**). Les messages sont ensuite filtrés pour être affichés ou non sur la console ou dans un fichier en fonction de la configuration des **appender** correspondants.
Rappelons que le logger racine (élément **root**) doit toujours exister dans le fichier de configuration **log4j.xml**

Maintenant, nous allons voir qu'il est tout à fait possible de "personnaliser" la journalisation en isolant les messages de journalisation d'une classe (ou d'un paquetage) dans un **logger** propre à cette classe (ou à ce paquetage).
Pour cela, il suffit de rajouter dans le fichier de configuration **log4j.xml**, un élément **logger** au-dessus de l'élément **root**.

→ L'attribut **name** de cet élément permet d'indiquer le nom du **logger**, liant ainsi le **logger** à une classe (ou un paquetage). Nous devons reprendre le nom du *logger* donné dans le programme Java qui correspond au paramètre d'entrée indiqué lors de l'appel à la méthode statique **getLogger**.
En effet, rappelez-vous qu'en début de tutoriel, nous avions instancié (au début de classe `PersonneDAOFichier`) un objet *logger* en appelant la méthode statique **getLogger** de la classe **Logger**.

```
private static Logger logger =
Logger.getLogger(PersonneDAOFichier.class.getName());
```

Comme la méthode *getLogger* prend en paramètre d'entrée le nom du *logger*, le *logger* pour la classe `PersonneDAOFichier` sera connu sous le nom :
com.iut.cabinet.metier.PersonneDAOFichier

Nous allons donc maintenant journaliser les traces de la classe **PersonneDAOFichier** dans un nouveau **logger** autre que le logger racine (**root**).

Rappel de la page 9 sur la configuration du `logger` dans un fichier de configuration **log4j.xml** :
La balise **level** permet d'indiquer le niveau minimum qu'un message doit avoir pour être pris en compte par le `Logger`. Elle est facultative, si elle n'est pas précisé, le niveau est hérité du parent .
→ La balise **appender-ref** permet d'indiquer dans l'attribut **ref** le nom d'un `Appender` déclaré précédemment. Il peut y avoir plusieurs balises **appender-ref** pour un `logger`.

Travail à faire :

→ **Préliminaire : supprimer la ligne rajoutée précédemment à propos du `threshold` dans la balise `appender` du `FileAppender`**

```
<appender name="fichier" class="org.apache.log4j.FileAppender">
    …
    <param name="threshold" value="warn"/>
    …
</appender>
```

→ **Pour ajouter un nouveau logger,** il suffit de rajouter une balise **`logger`** juste avant la balise **`root`** et juste après la balise **`appender`**, c-a-d que vous devez rajouter le code ci-dessous (disponible dans **`TestLog4j.txt`**) entre les balises **`</appender>`** et **`<root>`** du fichier **`log4j.xml`:**

```
<logger name="com.iut.cabinet.metier.PersonneDAOFichier" additivity="false">
<level value="DEBUG"/>
<appender-ref ref="console"/>
</logger>
```

→ La balise **`appender-ref`** indique que nous avons choisi d'envoyer les messages du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* dans la *`console`*. Pour visualiser la mise en place de ce nouveau log, nous allons supprimer pour le moment la *`console`* du logger racine (`root`). Ainsi les traces (messages) issues du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* seront orientées vers la console, et les autres (toutes les traces qui tombent dans le logger racine `root`) seront orientées vers le fichier.

```
<root>
    <level value="DEBUG"/>
<appender-ref ref="console"/>
    <appender-ref ref="fichier"/>
  </root>
```

Remarque : La mise en place d'un nouveau *logger* n'impacte bien sûr pas sur le nombre d'`appender` pouvant être associé à un élément `root` ou à un élément `logger`. Selon vos besoins, vous pourrez être amenés à paramétrer plusieurs `appender` dans un même `logger` ou dans un même `root`.

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que tous les messages de log (hormis ceux de niveau TRACE) sont affichés dans la console et qu'aucun message n'est affiché dans le fichier `cabinet.log`.
Cela signifie que les messages ont seulement bénéficié de l'**`appender`** du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* c-a-d de la console.

→ Il est possible que le logger *`com.iut.cabinet.metier.PersonneDAOFichier`* bénéficie à la fois de l'`appender` ***`console`*** qu'il référence, mais aussi de l'`appender` ***`fichier`*** que le logger racine **`root`** référence. Pour cela, il suffit de choisir une **`additivity`** à **`true`** lors du paramétrage du **`logger`** (nous l'avions déjà évoqué à la page 9). Les traces seront alors également récupérées par le logger "père" (ici `root`) .
Pour visualiser ce concept d'*additivité*, passer **`additivity`** à ***`true`*** :

```
<logger name="com.iut.cabinet.metier.PersonneDAOFichier" additivity="true">
```

Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que cette fois-ci, tous les messages de log (hormis ceux de niveau TRACE) sont affichés **à la fois** dans la console **et** dans le fichier `cabinet.log`.

→ Pour la suite de ce tutoriel, repasser l'***additivité*** à ***`false`***

```
<logger name="com.iut.cabinet.metier.PersonneDAOFichier" additivity="false">
```


Pour illustrer le fait que le *logger* racine (**`root`**) puisse récupérer des messages autres que ceux du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* (c.-à-d. autres que les messages de log issus de la classe `PersonneDAOFichier`), nous allons mettre en place des traces dans une autre classe utilisée par notre programme test, à savoir la classe **`PatientRegle`,** et plus particulièrement dans la méthode statique **`verifierNir`.**

Travail à faire :

→ **Mise en place d'une trace dans la méthode `verifierNir` de la classe `PatientRegle`**
Au tout début de la méthode **`verifierNir`**, écrire le message de log de niveau **`debug`** suivant :

```
logger.debug("Entrée dans la méthode verifierNir");
```

Une erreur de compilation apparaît... C'est normal ! Il faut instancier un *`logger`* au tout début de la classe `PatientRegle` de la manière suivante :

```
private static Logger logger = Logger.getLogger(PatientRegle.class.getName());
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que tous les messages du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* sont affichés dans la console et que les autres messages (du logger *`com.iut.cabinet.metier.PatientRegle)`* sont affichés dans le fichier `cabinet.log`.
Rappel : pour ce test, l'**`additivity`** du **`logger`** doit se trouver à **`false`** ...

Juste pour le plaisir, nous allons maintenant inverser les affichages des messages dans la console et le fichier, c-a-d que nous allons faire en sorte que tous les messages du logger *`com.iut.cabinet.metier.PatientRegle`* soient affichés dans la console (c-a-d "tombent" dans l'élément `logger` de **`log4j.xml`**) et que les autres messages (du logger *`com.iut.cabinet.metier.PersonneDAOFichier)`* soient affichés dans le fichier `cabinet.log` (c-a-d "tombent" dans l'élément `root` de **`log4j.xml`**)

→ Pour cela, modifier le `name` de l'élément `logger` du fichier **`log4j.xml`** et mettre maintenant le nom complet de la classe `PatientRegle`, à savoir : *`com.iut.cabinet.metier.PatientRegle`*

```
<logger name="com.iut.cabinet.metier.PatientRegle" additivity="false">
    …
</logger>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que seuls les messages du logger *`com.iut.cabinet.metier.PatientRegle`* sont désormais affichés dans la console (en début d'exécution, au moment où on passe dans le **`setNir`,** c-a-d avant l'affichage) et que les autres messages (du logger *`com.iut.cabinet.metier.PersonneDAOFichier)`* sont affichés dans le fichier `cabinet.log`.

Remarque A ce stade, nous pourrions envisager d'avoir dans le fichier **`log4j.xml`**, deux éléments `logger`: l'un ayant comme paramètre **`name`** *`"com.iut.cabinet.metier.PatientRegle"`* , l'autre ayant comme paramètre **`name`** *`"com.iut.cabinet.metier.PersonneDAOFichier"`*. Nous pourrions également envisager de déclarer plusieurs éléments `appender` de **`name`** : *`"fichier"`* pour mémoriser les traces de chaque `logger` dans des fichiers de noms différents … Rassurez-vous, pour l'instant, on ne vous demande pas de mettre en place ces nouveaux éléments `logger` et `appender` (s'il vous reste du courage à la fin du tutoriel, vous pourrez essayer de rajouter dans votre fichier de configuration autant d'éléments `appender` et/ou `logger` que vous le souhaitez )…

➢ Illustration par la pratique de la notion de hiérarchie appliquée aux loggers :

La remarque précédente nous laisse penser qu'il pourrait être possible de créer un élément `logger` dans le fichier **`log4j.xml`** pour chaque classe Java du projet. Mais dans ce cas, il y aurait beaucoup trop de `logger` à configurer et beaucoup trop de fichiers de log à consulter !
Dans cette partie, nous allons voir qu'il est en fait possible de configurer un seul élément **`logger`** du fichier de configuration **`log4j.xml`** pour toutes les traces issues des classes d'un même paquetage (voir d'un même projet).

Nous souhaitons que le logger *`com.iut.cabinet.metier.PersonneDAOFichier`* et le logger *`com.iut.cabinet.metier.PatientRegle`* aient les mêmes caractéristiques
(c-a-d le même comportement : même niveau de criticité et même `appender`), et que ces caractéristiques soient regroupées dans un seul élément *`logger`* du fichier de configuration **`log4j.xml`**.
Pour cela, il suffit de paramétrer le nom (paramètre **`name`**) de l' élément `logger` du fichier de configuration **`log4j.xml`** avec la partie commune des deux noms des deux logger c-a-d :

`com.iut.cabinet.metier`

(soit dit en passant, ce nom est également le nom du paquetage contenant les classes *`PersonneDAOFichier`* et *`PatientRegle`*)

Travail à faire :

→ **Illustration de la notion d'héritage de `logger` : Mise en place dans `log4j.xml` d'un logger commun aux classes du paquetage métier**

Raccourcir le `name` de l'élément `logger` du fichier **`log4j.xml`** et ne garder que le nom du paquetage métier, à savoir : *`com.iut.cabinet.metier`*

```
<logger name="com.iut.cabinet.metier" additivity="false">
    …
</logger>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que **tous les messages du logger** ***`com.iut.cabinet.metier`*** sont uniquement affichés **dans la console** (en début de console, on retrouve bien les traces issues du logger *`com.iut.cabinet.metier.PatientRegle`* enregistrées lors du passage dans la méthode `verifierNir` et en fin de console, on retrouve les traces issues du logger *`com.iut.cabinet.metier.PersonneDAOFichier`* enregistrées lors du passage dans la méthode `storeAllPersonnes`).
A ce stade, aucun message ne doit être affiché dans le fichier `cabinet.log`, puisque l'**`additivity`** est à `false`.

→ **Illustration de la notion d'additivité d'`appender` : Mise en place dans `log4j.xml` d'une additivité de logger**

Paramétrer maintenant l'`additivity` de l'élément `logger` à *`true`*.

```
<logger name="com.iut.cabinet.metier" additivity="true">
    …
</logger>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Vérifier que cette fois-ci, tous les messages **du logger** ***`com.iut.cabinet.metier`*** (hormis ceux de niveau `TRACE`) sont affichés **à la fois** dans la console **et** dans le fichier `cabinet.log`.
Remarque : *Le paramétrage choisis pour les messages de l'`appender` `File` du fichier `log4j.xml` à l'aide du pattern layout `ConversionPattern`, nous permet ici d'identifier aisément la provenance des traces présentes dans le fichier* ***`cabinet.log`***.

*Rappel de la page 9 de la définition de l'**`additivity`** donné lors la configuration du `logger` dans `log4j.xml` :* L'attribut **`additivity`** indique si l'additivité des `Appenders` s'applique ou non c-a-d si un logger d'un niveau donné va bénéficier de tous les Appenders de ces ancêtres en plus de ceux qui lui sont éventuellement affectés. Par défaut, ***l'additivité est active***.

A retenir sur l'`additivity` !
→ **Si l'`additivity` est à *`false`*,** on peut dire que le(les) **`appender`** associé(s) au **`logger`** a(ont) ***l'exclusivité de l'affichage des messages filtrés***
→ **Sinon l'`additivity` est à *`true`*,** et les messages filtrés sont transmis au logger père…

Remarque : La configuration au niveau des **`appender`** suit également une logique hiérarchique mais ce n'est pas de ***l'héritage mais une additivité***. Un **`appender`** définit dans un **`logger`** s'ajoute à ou aux **`appender`** déjà définis dans les loggers de la hiérarchie père.

⚠ **Additivité des `appender` ≠ hiérarchie des `logger`**

C'est pourquoi, nous allons dire quelques mots supplémentaires sur la hiérarchie de `logger` *…*

➢ Quelques mots supplémentaires sur la notion de hiérarchie pour les loggers …

Il faut savoir que l'API `log4j` gère les `logger` de façon hiérarchique, c'est à dire qu'un `logger` peut avoir des enfants et des parents.
La hiérarchie est basée sur le nom des `logger`. Le nom des `logger` est sensible à la casse.
Les niveaux sont définis par des **points,** comme la structure des **packages de classes**.
C'est pourquoi, lors l'instanciation d'un `logger` dans une classe Java, il est d'usage de le nommer avec le nom complet de la classe. C'est exactement ce que nous avons fait lors de l'appel de la méthode statique **`getLogger`**, où dans la classe `PersonneDAOFichier`, pour paramétrer le nom du `logger`, nous avons utilisé l'instruction : **`PersonneDAOFichier.class.getName()`**

Instancier un `logger` de nom **`com.iut.cabinet.metier.PersonneDAOFichier`** dans le programme Java, revient à mettre à disposition la hiérarchie de `logger` suivante :

```
com
   com.iut
        com.iut.cabinet
             com.iut.cabinet.metier
                  com.iut.cabinet.metier.PersonneDAOFichier
```

C'est-à-dire qu'il est possible de configurer le fichier `log4j.xml` avec un élément `logger` de nom **`com.iut.cabinet.metier.PersonneDAOFichier`**, et/ou avec un élément `logger` de nom **`com.iut.cabinet.metier`** et/ou avec un élément `logger` de nom **`com.iut.cabinet`** et/ou avec un élément `logger` de nom **`com.iut`** et/ou avec un élément `logger` de nom **`com.`**
On dit que :
- **`com.iut.cabinet.metier`** est le parent de **`com.iut.cabinet.metier.PersonneDAOFichier`**
- et que **`com.iut.cabinet`** est l'ancêtre de **`com.iut.cabinet.metier.PersonneDAOFichier`**

Heureusement, il est inutile de définir tous les `logger` dans le fichier de configuration `log4j.xml` puisque le principe d'héritage permet automatiquement à un `logger` d'obtenir les caractéristiques de son ascendant le plus proche pour lequel une configuration particulière a été précisée.

C'est bien ce que nous avons constaté précédemment, lorsque nous avons configuré le fichier **`log4j.xml`** uniquement avec le `logger` **`com.iut.cabinet.metier.`**
Les `logger` **`com.iut.cabinet.metier.PersonneDAOFichier`** et **`com.iut.cabinet.metier.PatientRegle,`** n'apparaissant plus explicitement dans la configuration **`log4j.xml,`** ont alors pris les caractéristiques de leur ascendant le plus proche c-a-d celles de l'élément `logger` nommé **`com.iut.cabinet.metier.`**

Explication : Lorsque l'on demande à l'API **`log4j`** de créer une instance de **`Logger`** dans le programme Java (appel de la méthode statique ***`getLogger`***) l'API **`log4j`** vérifie dans le fichier de configuration **`log4j.xml`** s'il existe un élément `logger` de même nom. Si elle ne trouve rien, elle remonte la hiérarchie des logger, recherchant dans le fichier de configuration **`log4j.xml`** l'élément `logger` ancêtre le plus proche afin d'obtenir ses caractéristiques et de le reporter sur le nouveau logger créé.
Si aucun élément `logger` n'est trouvé, c'est les caractéristiques de l'élément `root` qui seront reportées sur le nouveau logger créé. En effet, il faut savoir qu'au sommet de la hiérarchie des loggers, il y a toujours l'élément `root`… C'est pourquoi l'élément `root` qui est à la racine de la hiérarchie des loggers est obligatoire dans un fichier de configuration **`log4j.xml`**

→ **A propos du logger racine …**
`log4j` possède donc par défaut un `logger` à la racine de la hiérarchie. Il se nomme **`rootLogger`** dans l'API **`log4j`** et correspond à l'élément **`root`** du fichier de configuration **`log4j.xml`**
Pour obtenir une instance de ce logger racine, il faut utiliser la méthode `getRootLogger()` de la classe `Logger.` Le logger racine a deux caractéristiques par rapport aux autres loggers : c'est qu'il existe toujours et il n'a pas de nom.
De plus, dans le fichier **`log4j.xml`**, si aucun élément `logger` ne possède de niveau de criticité explicite (c-a-d si aucun `level` n'est explicitement écrit) dans la hiérarchie, c'est le `level` du logger racine (`root`) qui est utilisé. Et par défaut, le logger racine a un niveau de criticité égal à **`DEBUG`.**

A ce stade du tutoriel, dans notre projet, nous disposons pour les loggers de la hiérarchie suivante :

```
root
   com
      com.iut
         com.iut.cabinet
              com.iut.cabinet.metier
                   com.iut.cabinet.metier.PersonneDAOFichier
                   com.iut.cabinet.metier.PersonnePatientRegle
```


➢ **Optimisation des performances …**

Bien que **`log4j`** ait été développé pour réduire au minimum le surcoût de son utilisation, il est possible que l'utilisation de cette **`API`** influe sur les performances de votre application, même lorsque la journalisation est entièrement désactivée.

Prenons l'exemple suivant :

```
logger.debug("PI au carre:" + Math.PI * Math.PI);
```

Cette trace ne sera évidemment pas exécutée si le `logger` possède un niveau de criticité strictement supérieur à **`DEBUG`**. Néanmoins, le calcul engendré par les paramètres fournis à la méthode (`Math.PI * Math.PI`), ainsi que la concaténation chaîne-valeur, pourront entraîner une chute de performances.
En effet, le message `"PI au carre:" + Math.PI * Math.PI` a un coût de construction.

Afin de limiter le coût de construction du message, surtout si ce dernier doit être ignoré par le `logger`, la documentation officielle de l'API (voir partie **Performance** sur http://logging.apache.org/log4j/1.2/manual.html) conseille de **vérifier l'état de la journalisation** avant toute activité **à l'aide de la méthode `isDebugEnabled()`** (pour un message qui doit être loggué au niveau **`debug`**)

```
if (logger.isDebugEnabled()) {
      logger.debug("PI au carre:" + Math.PI * Math.PI);
   }
```

→ ***Avantage du test `if (logger.isDebugEnabled():`*** Le message ne sera construit que s'il est réellement pris en compte par le logger (c-a-d si niveau de criticité du *`logger`* est au minimum **`DEBUG`**)
→ ***"Inconvénient" du test `if (logger.isDebugEnabled():`*** Ce test est réalisé deux fois si le message est pris en compte par le logger : une fois par la méthode `isDebugEnabled()` et une autre fois par la méthode `debug()`. Cependant dans la plupart des cas, ce surcoût est beaucoup moins important que la création inutile du message.

Remarque : de la même manière, il existe les méthodes `isTraceEnabled`, `isInfoEnabled.`

Quelques conseils pour limiter les temps de traitement :
Utiliser un **`logger`** a nécessairement des impacts sur les performances de l'application. Cependant, si le **`logger`** est configuré judicieusement, ceux-ci sont généralement négligeables. Voici donc quelques conseils pour minimiser l'impact de **`log4j`** sur les performances de l'application :
→ Les temps de traitement de **`log4j`** dépendent de l'utilisation que vous ferez de **`log4j`** dans votre projet:
- Plus, il y aura de messages émis, plus les traitements seront longs : par exemple, il faut éviter d'envoyer un message dans une boucle
- Plus les niveaux de criticité associés à un `appender` seront bas dans la hiérarchie, plus le nombre de messages à traiter sera important
- Plus il y aura d'`appender`, plus le temps de traitement d'un message sera important

→ Attention, il est également bon à savoir que ***certains motifs de type `PatternLayout` sont connus pour être gourmand en temps de traitement***. Même si les informations de ces motifs sont particulièrement utiles, il faut tenir compte de leur temps de traitement lors de leur utilisation.
→ Lorsque vous avez besoin de concaténer des chaînes dans un `logger`, rappelez-vous que les `StringBuilder` sont moins coûteux que les `String`…
→ Enfin, pour économiser de la mémoire, il est également préférable de déclarer les **`logger`** en tant que variables statiques : **`private static Logger logger = Logger.getLogger(MaClasse.class);`**

Travail à faire : **Prise en compte de la méthode** `isDebugEnabled`

→ En début de méthode `verifierNir` de la classe `PatientRegle` compléter le message de log de la manière suivante :

```
if (logger.isDebugEnabled()) {
            logger.debug("Entrée dans la méthode verifierNir");
            }
```

→ En fin de méthode `verifierNir` de la classe `PatientRegle`, rajouter la trace suivante :

```
if (logger.isDebugEnabled()) {
            logger.debug("Sortie de la méthode verifierNir");
            }
```

→ En début de méthode `storeAllPersonnes` de la classe `PersonneDAOFichier`, commencer par effacer (ou commenter `/*…*/`), tous les messages de log utilisés pour les tests et rajouter la trace suivante :

```
if (logger.isDebugEnabled()) {
     logger.debug("Entree dans la méthode storeAllPersonne");
     }
```

→ En fin de méthode `storeAllPersonnes` de la classe `PersonneDAOFichier`, rajouter la trace :

```
if (logger.isDebugEnabled()) {
     logger.debug("Sortie de la méthode storeAllPersonne");
     }
```

→ Enregistrer et exécuter le code.

Vérifier que **les messages du logger *com.iut.cabinet.metier*** sont affichés **dans la console** (en début et en fin c-a-d au moment de l'exécution des méthodes `verifierNir` et `storeAllPersonnes`) **et dans le fichier `cabinet.log`**, puisque l'**`additivity`** est à **`true`**.

➢ *Choix du niveau de criticité des traces dans le programme Java*

A ce stade du tutoriel, la méthode **`storeAllPersonne`** doit être loguée en début et en fin.
Comme il s'agit de ***tracer une exécution***, ces logs doivent être réalisés avec le niveau **`DEBUG`**.
Votre code doit donc ressembler donc au code suivant :

```
public static void storeAllPersonnes (Collection<Personne> uneListe)
     {
     // Journalisation pour marquer l'entrée dans la méthode
     if (logger.isDebugEnabled())
            logger.debug("Entree dans la méthode storeAllPersonne");

     ////////////////////////////////////////////
     // ... Votre code reste ensuite inchangé ...
     ////////////////////////////////////////////

     // Journalisation pour marquer la sortie de la méthode
     if (logger.isDebugEnabled())
                    logger.debug("Sortie de la méthode storeAllPersonne");

      } // fin storeAllPersonnes
```


Travail à faire : **Méthode** `findAllPersonnes` :

→ De la même manière, vous loguerez en entrée et sortie la méthode **`findAllPersonnes`** de la classe **`PersonneDAOFichier`**

→ Enregistrer et exécuter le code.

Vous l'aurez compris, il est ***important de tracer le déroulement des applications en enregistrant les informations dans des journaux (fichiers logs) .***
Les contenus des logs sont **très variés** car ils rassemblent aussi bien des opérations de **fonctionnement normal** (tracé de l'exécution) que des **erreurs survenues**.

Jusqu'à présent, nous avons logué uniquement le fonctionnement normal de l'application (tracé de l'exécution), nous allons maintenant nous intéresser aux **erreurs** :
Dans notre application, nous allons trouver deux types d'erreurs :

- les **erreurs** liées au **métier** qui sont susceptibles de déclencher une **`CabinetMedicalException`**, ce qui est le cas lorsqu'un problème lié au métier apparaît : NIR incorrect, adresse du Patient absente … Nous décidons que dans notre projet, ces exceptions seront désormais logguées au niveau **`ERROR`**

- les **erreurs** liées à la **technique** qui déclencheront une **`CabinetTechniqueException`**. Ces erreurs sont des erreurs **`FATAL`**. Lorsque de telles erreurs se produisent en cours d'application :
  - l'erreur doit être remontée au niveau de l'IHM de l'utilisateur avec un message banalisé du genre :
    `Système en Erreur - Contacter l'administrateur`
    En effet, suite à ces erreurs, l'utilisateur ne peut rien faire, mieux vaut alors alerter un informaticien connaissant l'application, que nous appelons dans ce message administrateur (…alors que dans le cas d'une `CabinetMedicalException` liée par exemple à un NIR invalide, l'utilisateur pouvait re-saisir un NIR valide et l'application pouvait continuer)
  - Afin que l'administrateur puisse "réparer" l'application, il est indispensable que l'erreur soit logguée.

  Afin d'obtenir un log précis, nous décidons de rajouter dans le log les informations concernant l'exception déclenchée à l'aide de la méthode `getMessage()`.
  Ainsi lorsque l'administrateur interviendra, il consultera le log, qui d'une part contiendra le fonctionnement normal de l'application et d'autres part les informations liées à l'erreur technique.

Pour illustrer les **`CabinetTechniqueException`** , nous commencerons par nous intéresser à la méthode **`findAllPersonnes`**.

Au préalable : Récupérer sur la zone libre, la classe **`CabinetTechniqueException`** que vous importerez dans votre projet dans le paquetage **`com.iut.cabinet.metier`**

Travail à faire : Modifier la méthode de désérialisation `findAllPersonnes` de la classe `PersonneDAOFichier` de la manière suivante afin que lorsqu'une exception technique se déclenche (**`IOException`** ou **`ClassNotFoundException`**), on logue correctement l'erreur (**FATAL**) et on déclenche une **`CabinetTechniqueException`** pour informer l'utilisateur du problème technique :

```
public static Collection<Personne> findAllPersonnes () throws CabinetTechniqueException
      {

      if (logger.isDebugEnabled()) {
              { logger.debug("Entree dans la méthode findAllPersonne");  }


      Collection<Personne> maListe=null;
      try
      {
            FileInputStream fichier = new FileInputStream("cabMedPersonne.data");
            ObjectInputStream ois = new ObjectInputStream(fichier);
            maListe =(Collection<Personne>) ois.readObject();
          ois.close();
      }
      catch (IOException e)
            {
            // on logue aussi précisément que possible…
            logger.fatal("findAllPersonnes : Erreur Lecture dans le fichier "
                                                      + e.getMessage());

            // … puis on lance une exception technique
            throw new CabinetTechniqueException ("Système en Erreur - Contacter
l'administrateur");
            }

      catch(ClassNotFoundException e)
            {
            // on logue …
            logger.fatal("findAllPersonnes : Erreur de classe " + e.getMessage());

            // … puis on lance une exception technique
            throw new CabinetTechniqueException("Système en Erreur - Contacter
l'administrateur");
            }

      // … la fin de votre code reste inchangé …

}// fin findAllPersonnes
```

→ Exécuter (en ayant pris soin d'***attraper*** dans votre fichier de test la `CabinetTechniqueException`) et ouvrir le fichier `cabinet.log` généré …
Aucune trace de niveau **FATAL** n'est affichée…ce qui est normal puisque tout marchait bien avant de rajouter ces traces, il n'y a pas de raison pour que cela ne marche plus maintenant …

→ Pour produire une **`CabinetTechniqueException`**, enlever par exemple le **`implements Serializable`** dans la déclaration de la classe **`Personne.`**
Exécuter. Consulter la console et le fichier de **`cabinet.log`**, vous y retrouverez les traces **FATAL** concernant l'exception technique levée …

➢ Illustration par la pratique du choix du niveau de criticité `<level>` dans les loggers du fichier de configuration `log4j.xml`:

Travail à faire :

→ **Modification en FATAL du `level` de l'élément `root` :**
Modifier le `level` de l'élément `root` du fichier **`log4j.xml`** et le passer au niveau *`FATAL`* :

```
<root>
    <level value="FATAL"/>
    <appender-ref ref="fichier"/>
</root>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Consulter le contenu du fichier de log **`cabinet.log`** et constater que toutes les traces issues du log *`com.iut.cabinet.metier`* se retrouvent dans ce fichier (traces issues des classes *`com.iut.cabinet.metier.PatientRegle`* et *`com.iut.cabinet.metier.PersonneDAOFichier`*)

→ Pour bien comprendre le rôle du **`level`** de l'élément **`root`**, nous allons maintenant modifier le nom du logger et nous contenter de logger uniquement les traces issues de la classe **`PatientRegle`**.

```
<logger name="com.iut.cabinet.metier.PatientRegle" additivity="true">
    <level value="DEBUG"/>
    <appender-ref ref="console"/>
</logger>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.
Consulter le contenu du fichier de log `cabinet.log` et constater que cette fois-ci le fichier contient grâce à l'additivité (**`additivity`** à `true`), toutes les traces de niveau minimum **DEBUG** issues de l'élément **`logger`** (c-a-d de la classe **`PatientRegle`**) et toutes les traces de niveau minimum **FATAL** issues des autres logs (élément **`<root>`**) (c-a-d dans notre cas, uniquement les traces de niveau **FATAL** issues de la classe **`PersonneDAOFichier`**)

Nous avons précisé précédemment que lors d'erreurs fatales il fallait faire appel à l'administrateur.
Lorsque ce dernier intervient, il commence par consulter le fichier de log (`cabinet.log`).
Afin qu'il puisse facilement suivre les traces du projet en cours d'exécution, il est donc intéressant de paramétrer le fichier **`log4j.xml`** comme indiqué précédemment c-a-d avec un(des) élément(s) **`logger`** ayant une **`additivity`** à **`true`** et un élément **`root`** avec un **`level`** à **`FATAL`** (afin que seules les erreurs techniques extérieures au projet apparaissent éventuellement dans le fichier de log)…

➢ … Mais combien d'éléments `logger` doit-on paramétrer le fichier `log4j.xml` ?
Pour le projet `CabinetMedical` sur lequel nous sommes en train de travailler, nous vous proposons de paramétrer un seul élément **`logger`** pour tout le projet dans le fichier **`log4j.xml`**.
Ce logger doit donc être l'ancêtre de tous les loggers qui seront déclarés dans le fichier, c'est pourquoi on lui donnera le nom du projet, à savoir : ***`com.iut.cabinet`***

Travail à faire :
Mise en place de l'élément `logger` pour le projet du cabinetMedical `level` de l'élément `root` :
Modifier le nom de l'élément `logger` du fichier **`log4j.xml`** et lui donner le nom du projet

```
<logger name="com.iut.cabinet" additivity="true">
    <level value="DEBUG"/>
    <appender-ref ref="console"/>
</logger>
```

→ Enregistrer le fichier **`log4j.xml`** et exécuter le code.

Remarque :
Nous choisissons pour la suite de ne configurer qu'un seul élément **logger** pour notre projet.
Nous aurions pu effectuer d'autres choix de configuration et paramétrer par exemple dans le fichier de configuration, un élément **logger** par couche (paquetage) de notre projet, c-a-d que nous aurions pu avoir dans le fichier **log4j.xml** différents éléments **logger** de nom suivants : *com.iut.cabinet.metier, com.iut.cabinet.application, com.iut.cabinet.présentation, etc…* et nous aurions également pu choisir d'enregistrer les traces dans des fichiers de logs de noms différents…

➢ ***Pour information… Quelques mots sur les fichiers de journalisation dits "à rotation" :***
RollingFileAppender et DailyRollingAppender

Jusqu'à présent, nous avons travaillé avec la classe **FileAppender** qui permet de paramétrer un **appender** afin d'envoyer les traces dans un fichier de journalisation que nous avons appelé **cabinet.log.** Rappelons que :
- La propriété *"file"* permet de donner le nom souhaité au fichier de log
- le layout avec le paramètre *ConversionPattern* permet de formater les traces
- et bien sûr d'autres propriétés sont disponibles avec un **FileAppender** (voir la documentation en ligne de **log4j** sur : http://logging.apache.org/log4j/1.2/apidocs/index.html)

Mais un fichier de log peut très vite devenir volumineux…
En effet, la profusion de traces dans le fichier de **log** et/ou une mauvaise configuration peuvent impliquer un accroissement important de la taille du fichier : il est alors intéressant ***de pouvoir gérer la rotation des fichiers de logs, notamment en production***.
Ainsi, plutôt que de travailler avec un seul fichier de log dont la taille va rapidement grossir et le rendre inexploitable, on peut faire en sorte d'archiver plusieurs fichiers de logs selon différents critères (1 par jour, par semaine, nouveau fichier lorsque le fichier courant atteint une taille donnée,…).

Pour cela, l'API **log4j** propose deux classes héritées de **FileAppender** qui permettent de paramétrer un **appender** comme un fichier de journalisation " à rotation" : ce sont les classes **DailyRollingAppender** et **RollingFileAppender**.

↳ **La classe RollingFileAppender** permet d'envoyer les traces dans un fichier qui va "tourner" (c-a-d être archivé) lorsqu'il aura atteint une taille critique (fixée par la valeur de la propriété *maxFileSize*). Le fichier sera alors renommé pour être archivé et la journalisation reprendra dans un nouveau fichier **cabinet.log**

↳ **La classe DailyRollingAppender** permet d'envoyer les traces dans un ***fichier "à rotation périodique"***, c-a-d un fichier qui tourne régulièrement, mais contrairement à ce que son nom suggère, cette rotation n'est pas obligatoirement effectuée tous les jours.
C'est la propriété *DatePattern* qui va permettre de définir la périodicité de rotation. Par exemple, la valeur *'.'yyyy-MM-dd"* pour le *DatePattern* permet de mettre en place une rotation chaque jour (**dd**) à minuit.

Pour en savoir plus sur les fichiers de journalisation à rotation et les classes **RollingFileAppender** et **DailyRollingAppender**, rendez-vous à l'annexe 2 du tutoriel et consultez la documentation en ligne de l'API **log4j**.

… On ne vous demande pas pour l'instant de mettre en place de tels fichiers de log dits "à rotation"….

Conclusion :
L'utilisation d'un outil de ***journalisation*** tel que **log4j** permet d'inclure des messages (traces) à l'intérieur d'un code java et, de configurer ***au moment de l'exécution*** si les messages doivent être ou non logués sur la console ou dans des fichiers en fonction de leur niveau de gravité.

Comme nous venons de le voir dans ce tutoriel, **log4j** permet de gérer la journalisation de façon précise et adaptée via un fichier de configuration XML : **log4j.xml**
Par exemple, pour une phase de développement, le niveau des traces peut être fixée à un niveau bas **DEBUG**.
Puis, lors de la mise en production, il suffira de changer le niveau du log dans le fichier de configuration **log4j.xml** , le mettre par exemple à un niveau de criticité plus haut (comme ERROR) pour obtenir un journal de logs moins volumineux, tout en restant significatif.

➢ ***Retour sur l'intérêt du logging lors de la mise en production*** :
En production, un ou plusieurs fichiers de log au format prédéfini sont générés **en cours d'exécution** et conservent des messages informant sur la date et l'heure de l'événement, la nature de l'événement et sa gravité par un code ou une description sémantique, éventuellement d'autres informations : utilisateur, classe, etc…
Les fichiers logs d'une application représentent **la mémoire d'une application**, **un historique permanent** de la vie de celle-ci : la journalisation consiste à garder une trace des événements survenus dans une application. ***Les fichiers log sont donc utilisés lorsque l'application est mise en production***.
Les journaux peuvent alors être utilement réutilisés par :
- un développeur (administrateur) afin de détecter des défaillances et de corriger les bugs qui en sont responsables, il est plus facile de repérer la source d'une défaillance si le journal est dense en informations (fonctions appelées, valeurs des paramètres passés…)
- Un utilisateur peut utiliser un journal afin de revenir sur un crash et refaire les opérations qui auraient été perdues (transactions)

Au préalable :

Pour continuer, n'oubliez pas de remettre le **implements Serializable** dans la déclaration de la classe **Personne**.

Travail à faire : ***Tout Logguer !!!***

Comme expliqué précédemment, un code "de qualité" doit contenir des traces.
Vous devenez donc maintenant **reprendre tout le code de votre projet et le logguer correctement**…
A l'avenir, le code que vous écrirez devra également être loggué.
La notation des TP prendra bien évidemment en compte la mise en place des traces dans votre projet.

… Dorénavant, en vue de la mise en production de votre application et afin :
- *de suivre le bon déroulement de cette application : toutes vos méthodes devront être loguées en entrée et en sortie*
(hormis toString/hashcode/equals/getteurs/setteurs*(si pas de vérification de données… dès qu'il y aura une vérification dans le constructeur : il faudra bien sûr logguer)*
- *de pouvoir corriger les erreurs techniques qui pourraient survenir dans votre application, toutes vos exceptions techniques devront être loguées*

Dans le code Java, vous respecterez ce qui a été mis en place dans le tutoriel, c-a-d :

- trace de niveau **DEBUG** pour marquer **l'entrée et la sortie d'une méthode**
- trace de niveau **ERROR** pour marquer le déclenchement d'une **exception métier**
- trace de niveau **FATAL** pour marquer le déclenchement d'une **exception technique**

Dans le fichier de configuration `log4j.xml`, vous paramétrerez le niveau des `level` comme vous le souhaitez en fonction de vos besoins !

C'est bien ce que nous avons vu dans le tutoriel, suivant le paramétrage effectué dans le fichier de configuration, le journal de logs sera plus ou moins volumineux, tout en restant significatif.
Donc les niveaux paramétrés dans **`log4j.xml`** pendant une phase de développement seront certainement plus bas que ceux paramétrés pour la mise en production de l'application.
L'avantage d'utiliser un tel framework de journalisation et un fichier de configuration externe (**`log4j.xml`**) est qu'il est possible de gérer la présence ou non des traces comme vous le voulez, selon vos besoins, uniquement en intervenant dans le fichier **`log4j.xml`** sans retoucher une ligne du code Java !

Exemple de fichier de configuration : `log4j.xml`

A la fin de ce tutoriel, votre fichier de configuration **`log4j.xml`** doit ressembler au fichier de configuration proposé à l'annexe 3 de ce tutoriel.

Annexe 1 : Premier fichier de configuration `log4j.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration
xmlns:log4j="http://jakarta.apache.org/log4j/">
 <appender name="fichier" class="org.apache.log4j.FileAppender">
   <param name="file" value="./log/cabinet.log"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p]
%C %M [%L]: %m%n"/>
    </layout>
  </appender>
  <root>
    <level value="FATAL"/>
    <appender-ref ref="fichier"/>
  </root>
</log4j:configuration>
```

Annexe 2 : Fichiers de journalisation dits "à rotation" : `DailyRollingFileAppender` et `RollingFileAppender` et

La classe **`FileAppender`** permet de paramétrer un **`appender`** afin d'envoyer les traces dans un fichier de journalisation :

- le paramètre de `name "file"` permet de donner le nom souhaité au fichier de log
- le `layout` avec le paramètre *`ConversionPattern`* permet de formater les traces
- bien sûr d'autres propriétés sont disponibles avec un **`FileAppender`** (voir la documentation en ligne de **`log4j`** sur : http://logging.apache.org/log4j/1.2/apidocs/index.html)

Exemple configuration d'un appender avec *`FileAppender`*

```
<appender name="fichier" class="org.apache.log4j.FileAppender">
   <param name="file" value="./log/cabinet.log"/>
    <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p] %C %M [%L]:
%m%n"/>
    </layout>
</appender>
```

Pour paramétrer un **`appender`** comme un fichier de journalisation " à rotation", l'API **`log4j`** dispose de deux classes héritées de **`FileAppender`** : ce sont les classes **`DailyRollingAppender`** et **`RollingFileAppender`** .

La classe `DailyRollingAppender` permet d'envoyer les traces dans un ***fichier "à rotation périodique"***, c-a-d un fichier qui tourne régulièrement, mais contrairement à ce que son nom suggère, cette rotation n'est pas obligatoirement effectuée tous les jours.
C'est la propriété *`DatePattern`* qui va permettre de définir la périodicité de rotation et le suffixe des noms des fichiers créés à chaque rotation. La valeur de la propriété *`DatePattern`* suit le format utilisé par la classe **`SimpleDateFormat`.** Voici quelques exemples de valeurs pour la propriété *`DatePattern`* :

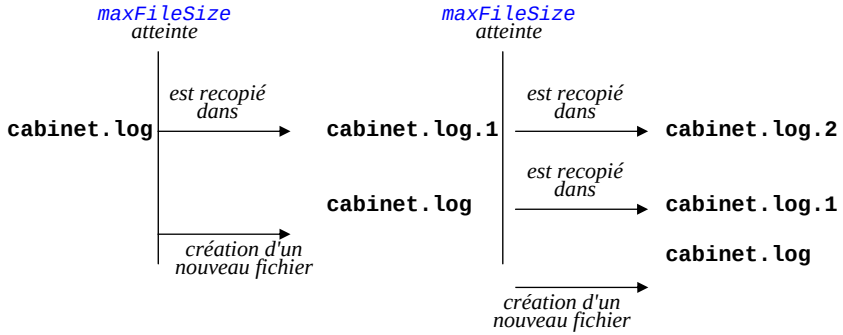
- **`'.'yyyy-MM`**: rotation chaque mois (**`MM`**)
- **`'.'yyyy-ww:`** rotation chaque semaine mois (**`ww`**)
- **`'.'yyyy-MM-dd:`** rotation chaque jour à minuit (**`dd`**)
- **`'.'yyyy-MM-dd-a:`** rotation chaque jour à midi et à minuit (**`dd-a`**)
- **`'.'yyyy-MM-dd-HH`**: rotation chaque heure (**`HH`**)

Exemple configuration d'un appender avec *`DailyRollingFileAppender`*

```
<appender name="LoggerFile" class="org.apache.log4j.DailyRollingFileAppender">
      <param name="file"  value="./log/cabinet.log"/>
        <param name="DatePattern" value="'.'yyyy-MM-dd" />
        <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p] %C %M [%L]:
%m%n"/>
      </layout>
 </appender>
```

Le choix de la valeur *`'.'yyyy-MM-dd"`* pour le *`DatePattern`* permet de mettre en place une rotation chaque jour (**`dd`**) à minuit. Par exemple, à minuit le 3 Octobre 2010, le fichier *`./log/cabinet.log`* sera renommé en fichier *`./log/cabinet.log.2010-03-10`* . Un nouveau fichier *`./log/cabinet.log`* sera créé, prêt à recevoir les traces du jour suivant : 11 Octobre 2010.

La classe `RollingFileAppender` permet d'envoyer les traces dans un fichier qui va "tourner" (c-a-d être archivé) lorsqu'il aura atteint une taille critique (fixée par la valeur de la propriété *`maxFileSize`*).
Lorsque la taille limite est atteinte le fichier est alors renommé pour être archivé et la journalisation reprend dans un nouveau fichier de log.
Autrement dit, le fichier **`cabinet.log`** est créé et rempli avec les différentes traces.
Une fois que la taille du fichier atteint la taille maximale (indiquée dans le fichier **`log4j.xml`** par la valeur de la propriété *`maxFileSize`*) , le fichier **`cabinet.log`** est renommé avec le suffixe **`.1`** c-a-d en **`cabinet.log.1`** et le fichier **`cabinet.log`** est recréé.
Une fois que le fichier **`cabinet.log`** est de nouveau rempli, le fichier **`cabinet.log.1`** est renommé en **`cabinet.log.2`**, et le fichier **`cabinet.log`** est renommé avec le suffixe **`.1`** en **`cabinet.log.1`** et un nouveau fichier **`cabinet.log`** est créé.



Remarque : Le nombre de fichiers conservés est quant à lui paramétré à l'aide de l'attribut *`maxBackupIndex.`*

Pour en savoir plus…*Documentation de l'API `log4j` en ligne sur :*

`http://logging.apache.org/log4j/1.2/apidocs/index.html`

Annexe 3 : Résumé : Feuille de route sur l'utilisation de `log4j` …

➢ *1. Mise en place de `log4j` dans votre projet*

→ **Créer un répertoire `lib`** où vous placerez le fichier jar de type **`log4j-x.x.x.jar`**
(le **`jar`** doit être inclus dans le **`classpath`**)

→ **Créer un répertoire `conf`** où vous placerez le fichier de configuration **`log4j.xml`**

→ **Créer un répertoire `log`** où vous stockerez le fichier de journalisation **`monFichier.log`**

➢ *2. Configuration de log4j via un fichier XML :*

Exemple de configuration pour un fichier `log4j.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
 <appender name="fichier" class="org.apache.log4j.FileAppender">
   <param name="file" value="./log/cabinet.log"/>
    <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{HH:mm:ss} [%-5p] %C %M [%L]:
%m%n"/>
    </layout>
  </appender>
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.SimpleLayout">
    </layout>
 </appender>
   <logger name="com.iut.cabinet" additivity="true">
    <level value="DEBUG"/>
    <appender-ref ref="console"/>
  </logger>
    <root>
    <level value="FATAL"/>
   <appender-ref ref="fichier"/>
  </root>
</log4j:configuration>
```

➢ *3. Utilisation de log4j dans les classes Java du projet :*

3.1 Instanciation d'un logger (statique) au début de chaque classe ayant comme nom, le nom de la classe :

```
private static Logger logger =
Logger.getLogger(PersonneDAOFichier.class.getName());
```

3.2 Mise en place des traces dans le code Java :

→ **Entrée et sortie des méthodes** (traces d'exécution ⇒ niveau **`DEBUG`**)

```
if (logger.isDebugEnabled())
            logger.debug("Entree dans la méthode storeAllPersonne");
```

→ **Exception métier** au niveau **`ERROR`**

→ **Exception technique** au niveau **`FATAL`**