

Jakarta Commons Configuration

par [Sébastien Le Ray](#)

Date de publication : 28/07/2005

Dernière mise à jour : 28/07/2005

Cet article décrit les mécanismes et l'utilisation de l'API Commons Configuration du projet Jakarta.

- I - Introduction
 - I.A - Contenu de l'API
 - I.B - Dépendances
- II - Configuration de base
- III - Configurations basées sur des fichiers
 - III.A - Fichier properties
 - III.A.1 - Fichiers properties classiques
 - III.A.2 - Fichiers properties XML
 - III.B - Fichiers XML
- IV - Autres types de configurations
 - IV.A - Base de données
 - IV.B - JNDI
 - IV.C - Bean Dynamique
 - IV.D - Configuration Système
- V - Configuration Composite
- VI - Configuration Dynamique
- VII - Décorateurs & Utilitaires
 - VII.A - Décorateurs
 - VII.A.1 - Support de types supplémentaires
 - VII.A.2 - Utilisation d'une Map comme configuration
 - VII.A.3 - Sous ensembles d'une configuration
 - VII.B - Utilitaires
 - VII.B.1 - ConfigurationUtils
 - VII.B.2 - Encapsulation d'autres configurations
- VIII - Conclusion
- IX - Remerciements
- X - Téléchargements

I - Introduction

I.A - Contenu de l'API

Comme son nom l'indique, l'API commons configuration de Jakarta offre un ensemble de classes permettant de gérer facilement la configuration d'une application. Les sources des données peuvent prendre la forme de classiques fichiers propriétés mais également de bases de données, de fichiers XML, de propriétés JNDI ou encore de propriétés système. Il est également possible de combiner différents types de configurations pour en offrir une vue unifiée. En outre, l'API offre un moyen pour préciser les types de configurations qui doivent être chargés au moyen d'un fichier XML.

Par ailleurs, un certain nombre de *wrappers* permettent de prendre en charge des configurations dans des cas particuliers comme celui des applets ou des servlets.

Toutes les classes de gestion de configuration implémentent l'interface **org.apache.commons.configuration.Configuration**

I.B - Dépendances

L'API commons configuration dépend de divers autres paquetages du projet Jakarta. Ces dépendances sont plus ou moins importantes selon les "modules" des commons configuration que l'on utilise. Tous ces modules seront présentés de façon plus détaillée dans la suite de cet article.

Le coeur de l'API commons configuration nécessite un environnement d'exécution Java 1.3 minimum ainsi que la présence sur le classpath des paquetages commons-collections et commons-lang. Il s'agit donc là des dépendances minimales pour pouvoir utiliser les commons configuration, elles suffisent pour une configuration basique (en mémoire ou à partir de propriétés systèmes).

Le paquetage commons logging est requis par un certain nombre de gestionnaires de configuration la **DataBaseConfiguration**, la **XMLConfiguration** et la **JNDIConfiguration** ainsi que par les classes **ConfigurationDynaBean** et **ConfigurationFactory**. Par ailleurs, il est également utilisé par la classe **org.apache.commons.configuration.ConfigurationUtils** ce qui implique que l'utilisation d'une **PropertiesConfiguration** (configuration à partir d'un fichier *.properties) ajoute la dépendance envers ce paquetage.

Si vous envisagez de stocker vos données de configuration dans une base de données (**DataBaseConfiguration**), vous ajoutez des dépendances envers l'API JDBC 3.0 (fournie avec le JRE 1.4 ou le paquetage `jdbc2_0-stdext.jar`).

Le stockage de données au format XML (**XMLConfiguration**) nécessite la disponibilité d'un JRE 1.4 ou des paquetages `xml-apis` et `xml-sec`.

L'utilisation de la configuration comme d'un *Bean Dynamique* (**ConfigurationDynaBean**) nécessite la disponibilité du paquetage commons beanutils.

Enfin, si vous souhaitez utiliser une configuration composite (c'est-à-dire formée de données issues de différentes sources) pour laquelle les différentes sources sont définies dans un fichier XML (**ConfigurationFactory**), vous aurez besoin des commons-digester ainsi que des APIs XML.

II - Configuration de base

Comme nous l'avons dit, toutes les classes servant à gérer une configuration implémentent **org.apache.commons.configuration.Configuration**. Les configurations sont basées sur un principe de paires clé/valeur. L'interface de base définit des méthodes communes permettant d'obtenir différents éléments à partir de la *Configuration*. Celle-ci permet de récupérer des objets (ou des types primitifs) des types suivants : **BigDecimal**, **BigInteger**, **Boolean/boolean**, **Byte/byte**, **Double/double**, **Float/float**, **Integer/int**, *List d'Object*, **Long/long**, **Short/short** et **String**. Chaque méthode est déclinée en deux versions :

- une version qui ne prend que la clé. Si la clé n'existe pas dans la *Configuration*, une **NoSuchElementException** est soulevée, si la valeur n'est pas du type demandé, une **ConversionException** est soulevée ;
- une version qui prend la clé et une valeur par défaut (d'un type primitif ou wrapper) si la clé n'existe pas. Si la valeur n'est pas du type demandé, une **ConversionException** est soulevée.

Si vous souhaitez récupérer un wrapper (**Boolean**, **Long**, *etc.*), il faut fournir une valeur par défaut qui soit de la classe du wrapper, sinon c'est une valeur de type primitif, s'il en existe un, qui est renvoyée. Une méthode générique existe également, il s'agit de **getProperty(String key)**, elle renvoie un objet sans se préoccuper de son type. Il faut savoir que dans le cas où la configuration est sauvegardée par programmation, les objets sont écrits sous forme de chaîne et chargés sous la même forme, il est donc nécessaire d'avoir une méthode permettant de régénérer l'objet à partir de la chaîne. Lorsque plusieurs valeurs correspondent à la même clé, les méthodes classiques ne renvoient que la première. Vous pouvez alors passer par la méthode **getList** qui renvoie une *List* contenant toutes les chaînes correspondant à la clé. La méthode **getProperty** peut également être utilisée, elle renvoie un objet qui sera une instance de **Collection** s'il existe plusieurs valeurs pour la clé. Les éléments de configuration de type chaîne peuvent faire référence les uns aux autres grâce à la séquence **#{autreChaîne}**. Dans ce cas, la séquence sera remplacée par le contenu de la propriété **autreChaîne**.

Ainsi, la lecture d'un élément de configuration aura l'aspect suivant :

```
Configuration c = new BaseConfiguration(); // On crée une configuration simple
try {
    boolean b1 = c.getBoolean("valeurbool"); // Type de base (nsee ou ce
possibles)
    boolean b2 = c.getBoolean("valeurbool", true); // Type de base (ce possible)
    Boolean b3 = c.getBoolean("valeurbool", Boolean.TRUE); // Wrapper (ce possible)
} catch (NoSuchElementException nsee) {
    log.fatal("La propriété valeurbool n'est pas définie !");
} catch (ConversionException ce) {
    log.fatal("La propriété valeurbool doit être un booléen !");
}
```

Une série de méthodes est également fournie de façon à gérer la configuration *via* la programmation. Ainsi, **clearPropertie(String clé)** supprime la propriété identifiée par **clé**, **addPropertie(String clé, Object val)** ajoute la propriété définie par la paire clé/valeur, si une propriété avec la même clé existait déjà, elles sont chaînées dans une **List** contrairement à **setProperty(String clé, Object val)** qui supprime toute valeur préexistante.

Nous allons maintenant voir les différents types de configuration proposés par l'API commons configuration.

III - Configurations basées sur des fichiers

Il existe deux types de configurations basées sur les fichiers : les fichiers properties et les fichiers XML. Quel que soit le type choisi, un certain nombre de fonctionnalités communes sont disponibles. Nous les décrivons ici avant d'explorer chaque type plus en détail.

Les configurations basées sur des fichiers implémentent l'interface **org.apache.commons.configuration.FileConfiguration**. Celle-ci définit des méthodes permettant de charger la configuration à partir d'un fichier (**File**), de l'emplacement du fichier ou d'une **URL**. S'il est impossible de charger le fichier, une **ConfigurationException** est soulevée.

Il est possible de forcer le rechargement au cours de l'exécution en appelant la méthode **load** ; invoquée sans argument, celle-ci rechargera le même fichier que lors de la création de l'objet. Mais il est plus intéressant d'utiliser un mécanisme proposé par l'API appelé *ReloadingStrategy*. Ce mécanisme détermine si le fichier doit être rechargé ou non. Lors d'un appel à l'une des méthodes **get**, la stratégie est interrogée pour savoir si le fichier doit être rechargé. Par défaut, le fichier n'est jamais rechargé (**InvariantReloadingStrategy**), mais il est possible de modifier ce comportement en appelant la méthode **setReloadingStrategy** et en lui passant une instance de **FileChangedReloadingStrategy** qui effectue un rechargement à chaque fois que le fichier est modifié. Vous n'aurez probablement pas à créer vos propres *ReloadingStrategies*.

Comme pour le chargement, il est possible de forcer la sauvegarde à n'importe quel moment en appelant la méthode **save** qui sauvegarde la totalité de la configuration à l'emplacement spécifié lors de sa création si on ne lui passe pas d'arguments. Là encore, une méthode de sauvegarde automatique est proposée, elle est désactivée par défaut. Vous pouvez l'activer en appelant **setAutoSave(true)**, dès lors, chaque modification, définition ou suppression de propriété entrainera la sauvegarde du fichier.

L'extrait de code suivant met en évidence ces deux mécanismes :

```
FileConfiguration reader = new PropertiesConfiguration("test.properties");
FileConfiguration writer = new PropertiesConfiguration("test.properties");
writer.setAutoSave(true);
reader.setReloadingStrategy(new FileChangedReloadingStrategy());
System.out.print("Test d'existence de la clé nom : ");
System.out.println(reader.containsKey("nom"));
System.out.println("Le writer définit nom à beuss...");
writer.setProperty("nom", "beuss");
System.out.print("Valeur de nom pour le reader : ");
System.out.println(reader.getString("nom"));
```

La sortie de cette portion de programme est la suivante (à la première exécution) :

```
Test d'existence de la clé nom : false
Le writer définit nom à beuss...
Valeur de nom pour le reader : beuss
```

On voit que la configuration est bien écrite et rechargée lorsque c'est nécessaire. Voici le contenu du fichier **test.properties** après l'exécution (il existait auparavant mais était vide) :

```
# written by PropertiesConfiguration
# Thu Jun 30 21:05:24 CEST 2005

nom = beuss
```

Comme on le voit, il s'agit d'un fichier de configuration classique. Nous reparlerons de son format plus loin.

Divers autres paramètres peuvent être définis pour les configurations basées sur des fichiers, parmi ceux-ci citons le répertoire de base (pour les chemins relatifs) et l'encodage du fichier.

III.A - Fichier properties

III.A.1 - Fichiers properties classiques

Ce type de fichiers est couramment utilisé en Java. La classe les prenant en charge est **org.apache.commons.configuration.PropertiesConfiguration**. Il s'agit de fichiers textes classiques pour lesquels les différentes propriétés sont définies sous la forme **clé=valeur**. Il est possible d'introduire des commentaires dans le fichier en faisant précéder la ligne d'un caractère dièse (#). Ces commentaires ne sont pas conservés lorsque le fichier est sauvegardé par le programme (ce devrait cependant être le cas dans la prochaine version de l'API). Il est possible de faire correspondre plusieurs valeurs à une même clé en les séparant par une virgule ou en définissant plusieurs fois la clé avec des valeurs différentes. Si une valeur est trop longue pour tenir sur une ligne, il faut terminer la ligne par un antislash (\) et poursuivre normalement (sans repréciser la clé) sur la ligne suivante.

Une fonctionnalité intéressante des fichiers properties est la possibilité d'inclure d'autres fichiers de configuration. Pour cela, il suffit de créer des valeurs dont la clé est **include**. La valeur est alors le chemin absolu ou relatif (au fichier de configuration courant) du fichier à inclure. Ce comportement est actif sauf dans le cas où la configuration n'est pas créée à partir d'un fichier (utilisation du constructeur par défaut).

Enfin, il est possible de définir un en-tête qui sera repris en commentaire au début du fichier de configuration. Pour cela, il suffit d'appeler la méthode **setHeader(String entete)**.

Traditionnellement, ces fichiers sont utilisés lorsque l'utilisateur configure directement l'application en modifiant le fichier properties à la main. Comme les commentaires ne sont pas conservés, il est déconseillé de les sauvegarder par programmation.

III.A.2 - Fichiers properties XML

Les fichiers properties XML sont différents des fichiers XML que nous allons voir plus tard. La classe les prenant en charge est **org.apache.commons.configuration.XMLPropertiesConfiguration**, son utilisation requiert que les API **commons digester**, **commons beanutils** et de lecture de flux XML soient disponibles dans le classpath. Il s'agit du format de fichier de configuration par Java 5, bien qu'il puisse être utilisé avec d'autres JRE.

Le fichier de configuration minimal est le suivant :

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
</properties>
```

Ensuite, chaque propriété est spécifiée au sein d'une balise **entry** :

```
<entry key="cle">Valeur</entry>
```

Il est possible d'inclure un commentaire avant les balises **<entry>** en utilisant la balise **<comment>**. Contrairement à la **PropertiesConfiguration**, ce commentaire, s'il est présent, est repris au début du fichier (ce n'est pas le cas des commentaires XML classiques).

III.B - Fichiers XML

Les fichiers XML classiques sont gérés *via* la classe **org.apache.commons.configuration.XMLConfiguration**. Notez qu'aucune validation n'est effectuée quant à l'adéquation à la DTD spécifiée dans le fichier. Vous pouvez parfaitement vous passer de DTD et utiliser un fichier ayant la forme suivante :

```
<properties>
  <listeners>
    <listener>
      <name>l1</name>
    </listener>
    <listener>
      <name>l2</name>
      <param>
        <name>p1</name>
        <value>1</value>
      </param>
    </listener>
  </listeners>
  <writers>
    <writer>
      <name>w1</name>
    </writer>
  </writers>
</properties>
```

Ce fichier sera correctement analysé par la **XMLConfiguration**. L'accès aux éléments se fait de façon simple, la clé d'un élément est en fait le "chemin" pour y accéder *sans tenir compte de l'élément racine*. Par exemple, pour accéder au nom du premier writer, on utilisera un appel du type :

```
String wnom = conf.getString("writers.writer.name"); // renvoie "w1"
```

Lorsque plusieurs éléments se trouvant au même niveau ont le même nom, il faut préciser un indice afin d'accéder aux propriétés de chacun. Ces indices vont de 0 à [nombre d'éléments - 1]. Par exemple, pour accéder au nom du premier listener :

```
String lnom = conf.getString("listeners.listener(0).name")
```

Bien sûr, les créateurs des commons configuration ont pensé à tout, et il existe une méthode permettant de savoir combien d'éléments d'un type donné sont présents. Il s'agit de la méthode **getMaxIndex(String cle)** :

```
conf.getMaxIndex("listeners.listener"); // Renvoie 1 dans notre exemple
```

Notez que cette méthode n'est définie que dans la superclasse **org.apache.commons.HierarchicalConfiguration**, vous ne pouvez donc plus utiliser l'interface générique si vous désirez y accéder.

L'accès aux attributs se fait d'une façon un peu particulière. En effet, pour ne pas qu'il y ait de conflit entre les attributs et les balises imbriquées, ils sont représentés sous une forme spéciale. Ainsi, en supposant que l'on modifie notre exemple pour avoir ceci :

```
<properties>
  <listeners>
    <listener name="l1" />
    <listener name="l2" alias="list2, listener2">
      <param name="p1" value="1"/>
    </listener>
  </listeners>
</properties>
```

L'accès au nom du premier listener se ferait désormais de la façon suivante :

```
String lnom = conf.getString("listeners.listener(0).[@name]")
```

Le nom de l'attribut a été préfixé par [**@** et suffixé par **]**. Ces deux constantes sont définies dans **org.apache.commons.configuration.ConfigurationKey** (**ATTRIBUTE_START** et **ATTRIBUTE_END**). Notez que la virgule est un caractère spécial qui permet d'attribuer plusieurs valeurs à un attribut, ainsi, l'accès au deuxième alias du second listener se fera de la façon suivante :

```
String lnom = conf.getString("listeners.listener(1).[@name](1)")
```

La méthode **getMaxIndex** fonctionne également pour les attributs.

Contrairement aux fichiers propriétés classiques, les éventuels commentaires sont conservés lors d'une modification et d'une sauvegarde du fichier. Cela permet de gérer la configuration *via* un programme tout en laissant la possibilité à l'utilisateur de modifier manuellement la configuration en incluant des commentaires en tant qu'aide-mémoire.

IV - Autres types de configurations

IV.A - Base de données

La gestion d'une configuration fondée sur une base de données est prise en charge par la classe **org.apache.commons.configuration.DatabaseConfiguration**. Son utilisation requiert la présence des commons logging et de l'API JDBC sur le classpath.

Le stockage des données de la configuration se fait dans une table de la forme **T_CONFIG([NomConfig,] Clé, Valeur)**. Le champ **NomConfig** est facultatif, néanmoins, il peut servir à faire cohabiter plusieurs configurations dans la même base. Les différents paramètres concernant la structure de la table sont passés au constructeur. Ce dernier prend également en paramètre la *DataSource* permettant d'accéder à la base de données. Ainsi, la création d'une instance de configuration utilisant la table ci-dessus ressemblera à cela :

```
DataSource ds = ...;
Configuration conf =
    new DatabaseConfiguration(ds, "T_CONFIG", "NomConfig", "Cle", "Valeur", "NomDeLaConfiguration");
```

Les troisième et dernier paramètres sont facultatifs dans le cas où l'on ne stocke qu'une seule configuration dans la table.

Notez que ce type de configuration n'intègre pas de système de cache, ce qui signifie que la base est interrogée à chaque demande de propriété. L'avantage est que l'application est toujours synchronisée avec la configuration, l'inconvénient est que, mal utilisé, ce type de configuration peut générer un trafic réseau important.

IV.B - JNDI

La récupération de propriétés à partir d'un service de nommage JNDI se fait au moyen de la classe **org.apache.commons.configuration.JNDIConfiguration**. Celle-ci est relativement simple d'utilisation, il suffit de passer au constructeur un *Context* et/ou un préfixe, les recherches s'effectuent dans ce contexte avec le préfixe spécifié.

Si aucun contexte n'est indiqué, un nouveau contexte vide (**InitialContext**) est créé. Notez que ce type de configuration est en lecture seule, il est possible de lire des propriétés, de les supprimer (localement) mais pas d'en ajouter ou d'en modifier (**addProperty** et **setProperty**) sous peine de soulever une **UnsupportedOperationException**.

IV.C - Bean Dynamique

La classe **org.apache.commons.configuration.beanutils.ConfigurationDynaBean** est en fait un wrapper pour une configuration permettant de l'utiliser comme un *DynaBean*. Un *DynaBean* implémente l'interface **org.apache.commons.beanutils.DynaBean** et permet de définir dynamiquement les propriétés d'un bean.

Son utilisation est relativement simple, il suffit de passer la configuration à traiter comme un *DynaBean* au constructeur :

```
DynaBean bean = new ConfigurationDynaBean(new XMLConfiguration("config.xml"));
```

La configuration est alors utilisable comme un *DynaBean*.

IV.D - Configuration Système

La classe **org.apache.commons.configuration.SystemConfiguration** permet d'utiliser les propriétés système (par exemple, passées à la JVM *via* l'option -D) comme données de configuration. Cette classe n'apporte aucune méthode spécifique et ne définit qu'un constructeur vide. Les propriétés ajoutées ou supprimées impactent également la *Map* récupéré *via* **System.getProperties()**.

V - Configuration Composite

En plus de gérer des configurations stockées sous des formes diverses, l'API commons configuration offre une fonctionnalité intéressante consistant à fournir une vue unique de plusieurs configurations. Autrement dit, il est possible d'agréger plusieurs configurations de sources différentes et de les manipuler comme si elles n'étaient issues que d'une source. Ce procédé est mis en oeuvre au moyen de la classe **org.apache.commons.configuration.CompositeConfiguration**.

Bien sûr, l'utilisation d'une configuration composite nécessite de renoncer aux spécificités des différentes configurations et de n'utiliser que celles définies dans l'interface **org.apache.commons.Configuration**. Néanmoins, l'ajout de configurations se faisant au coup par coup, il est possible de paramétrer chaque configuration (par exemple, activer le rechargement automatique d'une configuration basée sur un fichier) avant de l'ajouter à la vue.

La configuration obtenue est modifiable, cependant tout appel à la méthode **setProperty** ou **addProperty** est appliquée à la configuration passée au constructeur (applée configuration en mémoire ou "in memory configuration"). Ainsi dans l'exemple suivant :

```
CompositeConfiguration conf =
    new CompositeConfiguration(new PropertiesConfiguration("config.properties"));
conf.addConfiguration(new XMLConfiguration("config.xml"));
conf.addProperty("couleur", "FF0000");
```

C'est la *PropertyConfiguration* qui est modifiée par l'ajout de la propriété couleur. Lors d'un appel à **clearProperty** (et donc à **setProperty**), la propriété est supprimée de toutes les configurations. Lors de la demande d'une propriété, les différentes configurations sont interrogées dans l'ordre d'ajout (en commençant par la configuration en mémoire), la valeur renvoyée est celle stockée dans la première configuration qui contient la clé. Si l'on demande la liste des valeurs correspondant à une clé, toutes les configurations sont interrogées et chacune ajoute ses valeurs s'il y a lieu.

La classe **CompositeConfiguration** définit par ailleurs différentes méthodes permettant d'accéder aux configurations qu'elle contient ou de les modifier. Notez que lorsque vous accédez aux différentes configurations (via la méthode **getConfiguration**), la configuration en mémoire a l'indice zéro. Cette dernière est également accessible en utilisant la méthode **getInMemoryConfiguration**.

VI - Configuration Dynamique

La seconde possibilité intéressante offerte par l'API commons configuration est la possibilité de créer des configurations dynamiques. Cela signifie que les sources des différentes configurations ne sont pas spécifiées dans le code source mais dans un fichier XML. Ce fichier est analysé par la classe **org.apache.commons.configuration.ConfigurationFactory** qui va créer une *CompositeConfiguration* contenant toutes les configurations spécifiées dans le fichier. Voici un exemple de fichier de définition des sources de configuration :

```
config.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
  <override>
    <properties fileName="userprefs.properties" />
    <properties fileName="default.properties" />
  </override>
  <additional>
    <properties fileName="extensions.properties">
    <properties fileName="additionalexts.properties" optional="true">
  </additional>
</configuration>
```

Comme on le voit, ce fichier est composé de deux sections, nous y reviendrons plus tard. Chaque source de configuration est déclarée dans une balise qui lui est propre. La balise dépend du type de configuration.

Le chargement d'une configuration dynamique est assez simple, il suit le schéma suivant :

```
ConfigurationFactory factory = new ConfigurationFactory();
factory.setConfigurationURL(Main.class.getResource("/config.xml"));
Configuration conf = factory.getConfiguration();
```

Il existe quatre balises correspondant aux quatre configurations supportées :

- **properties** : indique que la configuration à créer est une **PropertiesConfiguration**, l'attribut **fileName** spécifie le fichier à utiliser ;
- **xml** ou **hierarchicalXml** : crée une configuration XML à partir du fichier spécifié par l'attribut **fileName**. L'utilisation de la première forme est préférable pour une question de cohérence (la classe **HierarchicalXMLConfiguration** est *deprecated* et a été remplacée par **XMLConfiguration**) ;
- **jndi** : demande à la *ConfigurationFactory* de créer une configuration JNDI ;
- **system** : crée une **SystemConfiguration**.

Toutes les balises concernant des configurations basées sur des fichiers acceptent l'attribut **optional** auquel doit être assignée une valeur booléenne (**true/false**, **yes/no**, **on/off**) indiquant si la configuration est optionnelle ou non. Par défaut, cet attribut est à **false** ce qui signifie que l'absence du fichier de configuration indiqué dans la balise **fileName** soulève une **ConfigurationException**.

Revenons maintenant aux deux parties composant le fichier. Celles-ci indiquent la façon dont sont gérées les collisions, c'est-à-dire le comportement à adopter lorsqu'une même clé est définie dans plusieurs configurations.

Pour les configurations déclarées dans la partie **override**, les propriétés définies dans les premiers fichiers écrasent les suivantes. Imaginons que **userprefs.properties** contienne une propriété appelée **gui.font.color**, si celle-ci est également définie dans **default.properties** seule la valeur déclarée dans **userprefs.properties** sera conservée. Cela permet par exemple de gérer les préférences utilisateurs, **userprefs** contient les préférences et **default** les valeurs par défaut, si une propriété n'est pas définie dans **userprefs**, c'est la valeur par défaut qui est prise, sinon, la valeur par défaut est tout simplement ignorée.

En ce qui concerne les configurations définies dans la section **additional**, elles sont toutes fusionnées en une seule configuration. Les propriétés en double sont combinées et il est possible d'y accéder *via* des indices (comme avec la **XMLConfiguration**). Si l'on dispose d'un premier fichier qui contient la valeur *bleu* pour la propriété **gui.font.color** et d'un autre qui contient la valeur *rouge*, **conf.getString("font.color(0)")** renverra *bleu* et **conf.getString("font.color(1)")** renverra *rouge*. Les balises de définition de configuration dans la section **additional** acceptent l'attribut **at** qui indique le point de raccordement dans la configuration additionnelle. Par exemple, si l'on dispose d'un fichier **windowprefs** et d'un fichier **menuprefs** contenant chacun la propriété **font**, on peut utiliser le fichier de configuration suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<configuration>
  <additional>
    <properties fileName="windowprefs.properties" at="gui.window">
    <properties fileName="menuprefs.properties" at="gui.menu">
  </additional>
</configuration>
```

De cette façon, les deux propriétés citées précédemment seront accessibles *via* les clés **gui.window.font** et **gui.menu.font**, respectivement. Cela permet de décomposer un fichier en plusieurs petits fichiers et d'en simplifier la syntaxe (les clés sont moins longues).

VII - Décorateurs & Utilitaires

VII.A - Décorateurs

VII.A.1 - Support de types supplémentaires

L'interface **Configuration** supporte la récupération d'un certain nombre de types de propriétés. Néanmoins, ce choix peut s'avérer insuffisant dans certains cas. Pour pallier à cela, il existe le décorateur **org.apache.commons.configuration.DataConfiguration**.

Tout d'abord, il permet d'obtenir des tableaux et des *List* pour tous les types supportés par l'interface **Configuration** (méthodes **getxxxList** et **getxxxArray**). Les tableaux sont des tableaux de types primitifs (lorsqu'il en existe un) tandis que les *List* contiennent des wrappers. Il existe deux exceptions : les méthodes sur les entiers, où la méthode renvoyant un tableau s'appelle **getIntArray** et celle renvoyant une *List* **getIntegerList**, et les méthodes sur les **String** qui sont absentes (**Configuration** supporte de base la récupération de tableau).

En plus de l'ajout de fonctionnalités au niveau des types existants, le décorateur **DataConfiguration** apporte la possibilité de récupérer de nouveaux types (tous disposent de méthodes pour récupérer des tableaux et des *List*) :

- **Calendar** ;
- **Color** ;
- **Date** ;
- **Locale** ;
- **URL**.

Comme on le voit, il s'agit de types utilisés de façon relativement courante au niveau de configurations.

S'agissant d'un décorateur, son utilisation est relativement classique :

```
DataConfiguration data = new DataConfiguration(new XMLConfiguration("config.xml"));
Locale loc = data.getLocale("app.locale");
```

Il est possible d'accéder à la configuration décorée *via* la méthode **getConfiguration**.

VII.A.2 - Utilisation d'une Map comme configuration

La classe **MapConfiguration** permet de décorer une **Map** pour la traiter comme une configuration. De cette façon, il est possible d'utiliser les utilitaires ainsi que les décorateurs sur cette **Map** de façon indirecte. Notez que cette classe n'est utile que si vous disposez de la **Map** auparavant. Si vous voulez créer une configuration qui n'est pas liée à une source particulière, utilisez la classe **org.apache.commons.configuration.BaseConfiguration**.

VII.A.3 - Sous ensembles d'une configuration

Un sous ensemble de configuration est l'ensemble des propriétés de cette configuration qui commencent par un préfixe donné (par exemple, "gui.window" pour avoir toutes les propriétés concernant les préférences d'affichage des fenêtres). Il est possible d'en obtenir un de deux façons :

- *via* la classe **org.apache.commons.configuration.SubsetConfiguration**, son constructeur prend une *Configuration*, un préfixe qui correspond au préfixe ajouté à toutes les clés lors de l'interrogation et,

éventuellement, un délimiteur qui correspond au délimiteur des différentes parties de la clé (concrètement, il sera inséré entre le préfixe et la clé demandée lors de l'interrogation de la *Configuration* décorée) ;

- via la méthode **subset**, celle-ci renvoie une **Configuration** (en fait une **SubsetConfiguration**) qui ne contient que les clés commençant par le préfixe indiqué, le séparateur utilisé est le point (.).

Ainsi, si l'on reprend l'exemple précédent, on utilisera une des deux méthodes suivantes :

SubsetConfiguration

```
Configuration windowConf = new SubsetConfiguration(globalConf, "gui.window", ".");
```

Méthode subset

```
Configuration windowConf = globalConf.subset("gui.window");
```

Si la configuration originale (**globalConf**) contenait la clé "**gui.window.color**", l'accès à la valeur correspondante se fait désormais en appelant **windowConf.getColor("color")**;

VII.B - Utilitaires

VII.B.1 - ConfigurationUtils

La classe **org.apache.commons.configuration.ConfigurationUtils** est une classe utilitaire (c'est-à-dire qu'elle ne dispose que de méthodes statiques et que son constructeur est privé). Elle offre diverses méthodes permettant d'agir sur les configurations :

- **append(Configuration source, Configuration cible)** : permet d'ajouter tout le contenu de la configuration source à la configuration cible. Si une clé de la source existe déjà dans la configuration cible, les valeurs de la configuration source sont ajoutées sans écraser celles qui existaient auparavant ;
- **copy(Configuration source, Configuration cible)** : copie la source vers la cible, si une clé de la source existe déjà dans la configuration cible, les valeurs correspondantes sont écrasées ;
- **dump(Configuration conf, PrintStream/PrintWriter out)** : envoie tout le contenu de la configuration sous la forme clé = valeur vers le **PrintStream** ou le **PrintWriter** spécifié (par exemple **System.out**) ;
- **toString(Configuration conf)** : renvoie une chaîne correspondant à la configuration passée en paramètre (forme clé = valeur).

D'autres méthodes utilitaires ne concernent pas spécialement les *Configurations* mais sont utilisées par les différentes classes pour obtenir des informations sur un fichier :

- **getFile** : renvoie, si possible, un objet **File** correspondant au fichier dont le répertoire de base et le nom sont passés en paramètres (les URL sont gérées) ;
- **getURL** : renvoie, si possible, un objet **URL** correspondant au fichier dont le répertoire de base et le nom sont passés en paramètres ;
- **locate** : recherche le fichier dont le nom et le répertoire sont passés en paramètres dans le répertoire personnel de l'utilisateur, le classpath courant et le classpath système, si le fichier est introuvable, renvoie **null**.

VII.B.2 - Encapsulation d'autres configurations

Pour conclure, signalons que l'API commons configuration propose divers wrappers autour de configurations d'objets web, il s'agit des classes se trouvant dans le paquetage **org.apache.commons.configuration.web**. Elles permettent d'encapsuler la configuration d'un **Applet**, d'une **Servlet**, d'un **ServletContext**, d'un **FilterConfig** ou

d'une **ServletRequest**. Grâce à ces classes, on peut gérer ce type de configuration comme une configuration Jakarta et profiter de toutes les fonctionnalités de l'API. Seule contrainte, les configurations ainsi créées sont en lecture seule, toute tentative d'ajout ou de suppression d'une propriété soulèvera une **UnsupportedOperationException**.

VIII - Conclusion

Vous en savez maintenant un peu plus sur l'API commons configuration. Essayez de l'appliquer à vos différents projets, vous verrez, elle devient vite indispensable. Bien que j'aie essayé de rendre cet article le plus complet possible, il peut rester des zones d'ombre ou des imprécisions, vous pouvez me contacter par MP pour avoir des précisions sur certains points (parcourez la javadoc avant) ou me signaler des erreurs.

IX - Remerciements

Merci à adiGuba & vedaer pour leur relecture, à GnuX pour ses corrections.

X - Téléchargements

- [L'API commons configuration](#) ;
- L'article au format [HTML zippé \(mirroir\)](#) ;
- L'article au format [PDF \(mirroir\)](#).