99. Apache Commons

Le projet Apache Commons est un ensemble de bibliothèques regroupant des utilitaires par thèmes.

Le projet est composé de trois parties :

- Commons Proper : les bibliothèques actives
- Commons Sandbox : les bibliothèques en cours de développement
- Commons Dormant : les bibliothèques inactives

La partie Commons Proper contient de nombreuses bibliothèques :

Bibliothèque	Rôle
BCEL	Byte Code Engineering Library : manipuler le bytecode et créer des fichiers .class
BeanUtils	Faciliter l'utilisation de l'API Reflection
Betwixt	Fonctionnalité de type OXM (mapping objet/XML)
BSF	Bean Scripting Framework : interface pour utiliser des langages de scripting (JSR-223)
Chain	Implémentation du motif de concept chaîne de responsabilités
CLI	Analyser les arguments fournis par la ligne de commandes
Codec	Proposer des algorithmes d'encodage/décodage (Base64, ...)
Collections	Fournir différentes collections
Compress	Utiliser des algorithmes de compression de fichiers comme zip, tar, ...
Configuration	Gérer des configurations provenant de différents formats
CSV	Gérer des données au format CSV (chaque valeur est séparée par une virgule)
Daemon	
DBCP	Fournir une implémentation de pool de connexions vers une base de données
DbUtils	Fournir des utilitaires concernant JDBC
Digester	Faciliter l'extraction de données d'un document XML
Discovery	Proposer un service de localisation de ressources
EL	Interpréteur pour l'Expression Language défini dans les spécifications de JSP 2.0
Email	Faciliter l'utilisation d'emails
Exec	Faciliter la gestion de l'exécution de processus externe
FileUpload	Faciliter la mise oeuvre de fonctionnalités de type upload de fichiers dans une webapp
Functor	
Imaging	Fournir des fonctionnalités de manipulation d'images
IO	Fournir des utilitaires pour les opérations de type entrée/sortie
JCI	

JCS	Fournir une solution de cache
Jelly	
Jexl	
JXPath	Permettre la manipulation de Java Beans en utilisant la syntaxe XPath
Lang	Fournir des utilitaires de base
Launcher	
Logging	Wrapper qui permet d'utiliser plusieurs implémentations d'API de logging
Math	
Modeler	Faciliter la création de Model MBeans respectant les spécifications JMX
Net	Fournir des utilitaires pour les opérations de type entrée/sortie
OGNL	
Pool	Fournir une solution de type pool d'objets
Primitives	
Proxy	Faciliter la création de proxys dynamiques
SCXML	
Transaction	
Validator	
VFS	

Le site du projet est à l'url : http://commons.apache.org/

Ce chapitre contient plusieurs sections :

99.1. Apache Commons Configuration

L'API Apache Commons Configuration propose une solution pour faciliter la gestion de la configuration d'une application en gérant ses paramètres.

Apache Commons Configuration propose de nombreuses fonctionnalités dont les principales sont :

- le chargement de la configuration à partir de différentes sources (Fichiers Properties, Fichier XML, Fichier INI de Windows, Fichiers Property list (plist), JNDI, JDBC, Propriétés système, ...)
- l'agrégation possible de différentes sources avec une priorisation de leur importance dans une hiérarchie
- un accès typé à la valeur d'une clé qui peut être simple ou multi-valeurs
- la modification et la persistance de la configuration
- la gestion d'événements lors du changement de la configuration

Indépendamment de la solution proposée par Apache Commons Configuration pour accéder et gérer une configuration, cette bibliothèque permet aussi d'agréger plusieurs sources différentes pour composer le contenu de la configuration. Cette agrégation est réalisée en utilisant les classes ConfigurationFactory et CompositeConfiguration.

99.1.1. L'interface Configuration

L'interface Configuration permet de manipuler ou modifier les éléments de la configuration de manière générique. Chaque classe qui encapsule une source de configuration implémente l'interface Configuration.

Un élément d'une configuration est une paire clé/valeur.

L'interface définit de nombreuses méthodes pour obtenir la valeur typée d'une clé. Les types primitifs ou objets supportés sont : boolean/Boolean, byte/Byte, double/Double, float/Float, int/Integer, long/Long, short, BigDecimal, BigInteger, String, String[] et List<Object>.

Le nom de ces méthodes commence par get suivi du type de retour de la donnée et elles attendent toute au moins un paramètre de type String qui contient le nom de la clé dont on veut obtenir la valeur. La méthode tente alors de retrouver la valeur de la clé dans la configuration et de la convertir dans le type retourné par la méthode. La plupart de ces méthodes possède une surcharge avec un paramètre supplémentaire permettant de préciser une valeur par défaut si la clé n'est pas trouvée dans la configuration.

Par défaut, si la clé n'est pas trouvée et que la valeur de retour est un objet, alors la valeur retournée sera null. Il est possible de changer ce comportement par défaut pour lever une exception de type NoSuchElementException en invoquant la méthode setThrowExceptionOnMissing() en lui passant la valeur true. Une exception de type ConversionException est levée si la valeur ne peut pas être convertie dans le type retournée par la

méthode.

Pour les types primitifs, une exception de type NoSuchElementException est levée.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import java.util.NoSuchElementException;
04.
05. import org.apache.commons.configuration.BaseConfiguration;
06. import org.apache.commons.configuration.Configuration;
07.
08. public class TestBasicConfiguration {
09.
10.   public static void main(final String[] args) {
11.     final Configuration config = new BaseConfiguration();
12.     try {
13.       System.out.println(config.getBoolean("maValeur"));
14.
15.     } catch (final NoSuchElementException nsee) {
16.       System.out.println("La propriété maValeur n'est pas trouvée");
17.     }
18.     System.out.println(config.getBoolean("maValeur", true));
19.     System.out.println(config.getBoolean("maValeur", Boolean.TRUE));
20.     config.setProperty("maValeur", false);
21.     System.out.println(config.getBoolean("maValeur", true));
22.   }
23. }
```

Résultat :

```
1. La propriété maValeur n'est pas trouvée
2. true
3. true
4. false
```

La méthode getProperty(), qui attend en paramètre le nom de la clé, renvoie la valeur sous la forme d'un Object.

Les méthodes getList() et getStringArray() de l'interface Configuration, qui attendent en paramètre le nom de clé, permettent d'obtenir les valeurs associées à une propriété multi-valeurs. Une telle propriété est de type String et contient au moins un caractère qui est définit comme étant le délimiteur. Par défaut, ce caractère de délimitation de chaque occurrence est la virgule. La méthode setListDelimiter() permet de préciser un autre caractère.

Il est possible d'utiliser la méthode statique setDefaultListDelimiter() de la classe AbstractConfiguration pour modifier le caractère de délimitation pour toutes les configurations.

La méthode setDelimiterParsingDisabled() permet de désactiver le découpage des valeurs des propriétés en lui passant en paramètre la valeur true : dans ce, la configuration ne propose plus le support des multi-valeurs.

Les méthodes getList() et getStringArray() renvoie toujours null si la clé n'est pas trouvée.

La valeur d'une clé peut faire référence à une autre clé : lors de la récupération de la valeur la référence est remplacée par la valeur de la clé correspondante.

Plusieurs méthodes permettent de gérer le contenu de la configuration :

Méthode	Rôle
clearProperty(String key)	Supprimer la clé de la configuration
addProperty(String key, Object value)	Ajouter une clé dans la configuration en lui associant la valeur fournie. Si la propriété existe déjà, la nouvelle valeur est ajoutée à la valeur existante. La clé possède alors plusieurs valeurs
setProperty(String key, Object value)	Modifier la valeur d'une clé avec celle fournie en paramètre. La clé est créée dans la configuration si elle n'existe pas
clear()	Supprimer tous les éléments de la configuration

99.1.2. Les différents types de configuration

Toutes les classes qui encapsulent une source de configuration implémentent l'interface org.apache.commons.configuration.Configuration.

L'API propose une classe pour gérer chaque source de configuration :

- BaseConfiguration : la configuration est stockée en mémoire
- PropertiesConfiguration : la configuration est stockée dans un fichier de type .properties

- PropertyListConfiguration : la configuration est stockée dans un fichier de type .plist
- XMLPropertiesConfiguration
- XMLPropertyListConfiguration (Mac OS X)
- XMLConfiguration : la configuration est stockée dans un fichier XML
- INIConfiguration : la configuration est stockée dans un fichier de type .ini (Windows)
- SystemConfiguration : la configuration contient les variables d'environnement système
- DataBaseConfiguration
- XMLConfiguration
- JNDIConfiguration : les éléments de la configuration sont retrouvés en utilisant JNDI
- HierarchicalConfiguration : la configuration est stockée en mémoire
- ConfigurationDynaBean : permet d'utiliser la configuration comme un bean dynamique
- ConfigurationFactory : permet de gérer une configuration composite qui va pouvoir contenir plusieurs sources potentiellement de types différents.

En fonction des sources de configuration, l'API Common Configuration va requérir diverses dépendances.

99.1.2.1. Les configurations reposant sur des fichiers

Les classes qui encapsulent des configurations stockées dans des fichiers implémentent l'interface org.apache.commons.configuration.FileConfiguration.

L'interface FileConfiguration propose plusieurs méthodes pour préciser le fichier contenant la configuration.

- setFile() : attend en paramètre un objet de type java.io.File
- setURL() : attend en paramètre un objet de type URL
- setFileName() : attend un objet de type String qui précise le nom du fichier
- setBasePath() : attend un objet de type String qui précise le chemin du fichier

Une instance de type File ou une URL permet d'identifier de manière unique un fichier. Si le fichier est précisé en utilisant un chemin et/ou un nom de fichier alors le framework tente de déterminer le fichier dans un ordre précis :

- si la combinaison du chemin et du nom du fichier est un URL, alors cette URL est utilisée pour charger le fichier
- si la combinaison du chemin et du nom du fichier est un chemin absolu vers le fichier, alors ce chemin est utilisé pour charger le fichier
- si la combinaison du chemin et du nom du fichier est un chemin relatif vers le fichier, alors ce chemin est utilisé pour charger le fichier
- si le nom du fichier existe dans le répertoire HOME de l'utilisateur, alors ce fichier est chargé
- le fichier est considéré comme une ressource et sera chargé par un classloader à partir de son nom (le fichier doit dans ce cas être dans le classpath)

Si toutes ces tentatives échouent alors une exception de type ConfigurationException est levée.

La méthode load() permet de charger la configuration ou de lever une exception de type ConfigurationException si le chargement échoue. La méthode load() possède plusieurs surcharges qui permettent de préciser une source à charger différente de celle configurée dans l'objet (la source fournie en paramètre ne remplace pas la source configurée dans l'objet).

Il est possible d'invoquer plusieurs fois la méthode load() avec des sources différentes : dans ce cas, la configuration existante n'est pas réinitialisée mais elle est ajoutée.

Généralement les classes qui implémentent l'interface FileConfiguration proposent des constructeurs qui attendent en paramètre la source sous différents types et invoquent la méthode load() sur la source.

Commons Configuration propose le concept de stratégie de rechargement (reloading strategy). Il est ainsi possible d'utiliser un mécanisme nommé ReloadingStrategy qui permet de déterminer si la configuration doit être rechargée automatiquement si celle-ci a été modifiée. Cette stratégie est invoquée à chaque fois que l'une des méthodes getXXX() est appelée.

Par défaut, un FileConfiguration est associé à une stratégie InvariantReloadingStrategy qui précise que la configuration n'est jamais rechargée.

La méthode setReloadingStrategy() permet de préciser une instance de stratégie différente.

Le rechargement automatique d'une configuration dont la source a été modifiée est particulièrement intéressant notamment pour des applications qui doivent avoir un fort taux de disponibilité. Ce rechargement permet d'éviter d'avoir à redémarrer l'application pour obtenir les nouvelles valeurs de la configuration.

Si la source est un fichier, la stratégie de base est implémentée dans la classe FileChangedReloadingStrategy qui recharge le contenu du fichier si celui-ci a été modifié.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.ConfigurationException;
04. import org.apache.commons.configuration.XMLConfiguration;
05. import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;
06.
07. public class TestXMLConfiguration {
08.
09.   public static void main(final String[] args) {
10.     try {
11.       final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
12.       final FileChangedReloadingStrategy strategy = new FileChangedReloadingStrategy();
13.       strategy.setRefreshDelay(5000);
14.       config.setReloadingStrategy(strategy);
```

```
      System.out.println(config.getFile());
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

La stratégie FileChangedReloadingStrategy implique que la classe est invoquée à chaque accès à une propriété de la configuration pour vérifier si la date de dernière modification du fichier à changer depuis la précédente modification. Si la date a changée alors la configuration est rechargée. Pour éviter un accès disque lors de l'obtention d'une valeur de la configuration, il est possible de préciser un délai d'attente en millisecondes entre chaque vérification (refresh delay).

La classe ManagedReloadingStrategy est une alternative au rechargement automatique de la configuration : elle permet de forcer le rafraichissement à la demande en invoquant sa méthode refresh().

Exemple en utilisant Spring et JMX

```
<bean id="configuration" class=" org.apache.commons.configuration.PropertiesConfiguration">
  <constructor-arg type="java.net.URL" value="file:${user.home}/monAppConfig.properties"/>
  <property name="reloadingStrategy" ref="reloadingStrategy"/>
</bean>

<bean id="reloadingStrategy"
  class=" org.apache.commons.configuration.reloading.ManagedReloadingStrategy"/>

<bean id="mbeanMetadataExporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="server" ref="mbeanServer"/>
  <property name="beans">
    <map>
      <entry key="monApp:bean=configuration" value-ref="reloadingStrategy"/>
    </map>
  </property>
</bean>
```

La méthode save() permet d'enregistrer la configuration en remplaçant la source encapsulée dans l'objet. Il est possible de préciser un autre fichier cible en utilisant une des surcharges de la méthode save().

La méthode setAutoSave() attend en paramètre un booléen qui s'il vaut true permet de demander l'enregistrement de la configuration à chaque fois qu'elle est modifiée. Dans ce cas, chaque modification faite dans la configuration en utilisant l'API va sauvegarder la configuration. Attention toutefois si le nombre de mises à jour est important que cela peut engendrer beaucoup d'accès sur le système de fichiers.

Les changements vont être vérifiés périodiquement par la stratégie de rechargement (reloading strategy) associée à la configuration.

La méthode clear() permet de réinitialiser le contenu de la configuration.

99.1.2.1.1. Les fichiers properties

Ce format de fichier est supporté nativement par la plate-forme Java. Il permet de définir des paires clé/valeur dans un fichier texte, une sur chaque ligne, la clé étant séparée de sa valeur par un caractère '='.

La valeur peut contenir plusieurs valeurs qui doivent dans ce cas être séparées par un caractère ','. Il est aussi possible de définir plusieurs fois la clé en lui affectant à chaque fois une des valeurs.

Si la valeur doit tenir sur plusieurs lignes, chaque ligne sauf la dernière doit être terminée par un caractère '\' (antislash).

Les lignes commençant un caractère '# ' sont des commentaires et sont ignorées.

La classe org.apache.commons.configuration.PropertiesConfiguration encapsule une configuration dont la source est un fichier .properties.

Il est possible d'inclure le contenu d'autres fichiers .properties en utilisant comme clé 'include' et comme valeur le chemin absolu ou relatif du fichier .properties à inclure.

Les commentaires contenus dans le fichier .properties source sont perdus lors de l'enregistrement de la configuration. La méthode setHeader() permet cependant de préciser un commentaire qui sera écrit dans le fichier .properties lors de son enregistrement.

Plusieurs implémentations de l'interface Configuration héritent de la classe AbstractConfiguration.

Il faut créer un fichier .properties qui va contenir les éléments de la configuration.

Résultat :

```
data.database.url=127.0.0.1
data.database.port=3306
data.database.login=admin
data.database.password=password
```

Le plus simple est de créer une instance de la classe PropertiesConfiguration en lui passant en paramètre le nom du fichier .properties.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.Configuration;
04. import org.apache.commons.configuration.ConfigurationException;
05. import org.apache.commons.configuration.PropertiesConfiguration;
06.
07. public class TestPropertiesConfiguration {
08.
09.   public static void main(final String[] args) {
10.     try {
11.       final Configuration config = new PropertiesConfiguration("maConfig.properties");
12.       final String url = config.getString("data.database.url");
13.       System.out.println("url = " + url);
14.     } catch (final ConfigurationException e) {
15.       e.printStackTrace();
16.     }
17.   }
18. }
```

Si le chemin fourni en paramètre n'est pas un chemin absolu, alors le fichier est successivement recherché dans le répertoire courant, le répertoire HOME de l'utilisateur et dans le classpath.

Le fichier peut aussi être chargé en utilisant une des surcharges de la méthode load().

Il est possible de demander l'inclusion d'autres fichiers .properties en utilisant une paire clé/valeur particulière dans le fichier .properties : le nom de la clé doit être obligatoirement include et la valeur est le nom du fichier .properties à inclure.

Résultat :

```
1. include = db.properties
```

Il est possible d'associer plusieurs valeurs à une même clé en utilisant deux syntaxes :

- séparer les différentes valeurs associées à une clé en utilisant le caractère virgule
- définir une ligne pour chaque valeur avec la même clé

Résultat :

```
1. valeurs.paires = 0, 2, 4, 6, 8
2. valeurs.impaires = 1
3. valeurs.impaires = 3
4. valeurs.impaires = 5
5. valeurs.impaires = 7
6. valeurs.impaires = 9
```

Il est possible d'obtenir directement un tableau ou une collection des valeurs associées à la clé en utilisant les méthodes getStringArray() ou getList().

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import java.util.Arrays;
04. import java.util.List;
05.
06. import org.apache.commons.configuration.Configuration;
07. import org.apache.commons.configuration.ConfigurationException;
08. import org.apache.commons.configuration.PropertiesConfiguration;
09.
10. public class TestPropertiesConfiguration {
11.
12.   public static void main(final String[] args) {
13.     try {
14.       final Configuration config = new PropertiesConfiguration("maConfig.properties");
15.       final String[] valeursPaires = config.getStringArray("valeurs.paires");
16.       final List<Object> valeursImpaires = config.getList("valeurs.impaires");
17.       System.out.println("Valeurs paires=" + Arrays.deepToString(valeursPaires));
18.       System.out.println("Valeurs impaires=" + valeursImpaires);
19.     } catch (final ConfigurationException e) {
20.       e.printStackTrace();
21.     }
22.   }
23. }
```

Résultat :

```
Valeurs paires=[0, 2, 4, 6, 8]
Valeurs impaires=[1, 3, 5, 7, 9]
```

Pour sauvegarder une configuration modifiée, il suffit d'invoquer la méthode save().

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final PropertiesConfiguration config =
        new PropertiesConfiguration("maConfig.properties");
      System.out.println(config.getFile());
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      config.save();
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Attention : il est possible que le fichier utilisé lors de la sauvegarde ne soit pas celui utilisé pour lire la configuration si c'est uniquement le nom du fichier qui est fourni en paramètre du constructeur. La méthode getFile() permet de connaitre le chemin du fichier utilisé.

Il est possible de fournir en paramètre de la méthode save() le nom du fichier pour par exemple écrire dans un autre fichier.

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestPropertiesConfiguration {

  public static void main(final String[] args) {
    try {
      final PropertiesConfiguration config =
        new PropertiesConfiguration("maConfig.properties");
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      config.save("maConfig.properties.bck");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Certains caractères doivent être échappés dans le fichier properties :

- le retour chariot : \n
- la tabulation : \t
- le caractère antislash : \\
- les caractères Unicode : \u20ac
- le séparateur des éléments de liste de valeurs (virgule par défaut) : \,

L'échappement des caractères peut devenir délicat dans certaines circonstances.

Exemple :

```
config.sources = c:\\config\\\\,c:\\data\\
config.urls = \\\\\\\\host1,\\\\\\\\host2
```

Dans ce cas, il est plus lisible de définir chacune des valeurs sur une ligne dédiée.

Résultat :

```
config.sources = c:\\config\\
```

```
config.sources = c:\\data\\
config.urls = \\\\host1
config.urls = \\\\host2
```

Une instance de type PropertiesConfigurationLayout est associée aux objets de type PropertiesConfiguration. Le rôle de la classe PropertiesConfigurationLayout est de préserver autant que possible la structure originale du fichier de properties qui a été chargé lorsque la configuration est sauvegardée. Par exemple, elle tente de conserver les commentaires et les lignes blanches.

La configuration par défaut de cet objet tente de préserver au mieux la structure du fichier .properties : plusieurs propriétés peuvent être modifiées pour des besoins spécifiques.

99.1.2.1.2. Les fichiers properties au format XML

Depuis la version 5.0 de Java, il est possible de stocker les paires clé/valeur encapsulées dans une instance de type properties dans un fichier XML en utilisant la méthode loadFromXML().

La classe XMLPropertiesConfiguration propose un support d'un fichier XML contenant les données d'un Properties comme source de données.

Dans cas, le format du fichier XML est imposé par les spécifications de l'API Java dans une DTD

Résultat :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for properties -->
<!ELEMENT properties ( comment?, entry* ) >
<!ATTLIST properties version CDATA #FIXED "1.0">
<!ELEMENT comment (#PCDATA) >
<!ELEMENT entry (#PCDATA) >
<!ATTLIST entry key CDATA #REQUIRED>
```

Le fichier XML minimal est donc

Exemple :

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

Chaque paire de clé/valeur est définie dans un tag <entry>. Le nom de la clé est fourni dans l'attribut key et la valeur est fournie dans le corps du tag.

Exemple :

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM «http://java.sun.com/dtd/properties.dtd»>
<properties>
  <entry key="cle">Valeur</entry>
</properties>
```

Les commentaires sont définis en utilisant le tag <comment>.

L'utilisation de la classe XMLPropertiesConfiguration requiert les bibliothèques Commons Digester, Commons BeanUtils et un parser XML comme dépendances.

99.1.2.1.3. Les fichiers XML

La classe org.apache.commons.configuration.XMLConfiguration encapsule une configuration dont la source est un fichier XML.

La structure du fichier XML est libre : elle n'est pas vérifiée à partir d'une DTD ou d'un schéma XML. Le nom des clés est déterminé en utilisant le nom chaque tag père (sauf le tag racine) et le nom du tag lui-même séparés par un caractère point.

L'utilisation de la classe XMLConfiguration requiert deux dépendances du projet apache Commons : BeanUtils et JxPath.

Exemple avec Maven :

```xml
<dependencies>
    <dependency>
        <groupId>commons-configuration</groupId>
        <artifactId>commons-configuration</artifactId>
        <version>1.8</version>
    </dependency>
    <dependency>
        <groupId>commons-beanutils</groupId>
        <artifactId>commons-beanutils</artifactId>
        <version>1.8.0</version>
    </dependency>
    <dependency>
        <groupId>commons-jxpath</groupId>
        <artifactId>commons-jxpath</artifactId>
        <version>1.3</version>
    </dependency>
</dependencies>
```

La classe XMLConfiguration lit la configuration à partir d'un fichier XML et mappe la hiérarchie des tags sur la notation basée sur les points.

Par exemple : <config><database><url> est mappé sur database.url

Exemple :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <database>
    <url>127.0.0.1</url>
    <port>3306</port>
    <login>admin</login>
    <password>password</password>
  </database>
</config>
```

L'utilisation de ce fichier de configuration requière la création d'une instance de type XMLConfiguration et l'utilisation de la notation par points pour obtenir les valeurs.

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
String url = config.getString("database.url");
String port = config.getString("database.port");
```

Il est possible d'accéder à un élément fils possédant plusieurs occurrences en utilisant son index.

Exemple :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <databases>
    <database>
      <env>dev</env>
      <url>127.0.0.1</url>
      <port>3306</port>
      <login>admin</login>
      <password>password</password>
    </database>
    <database>
      <env>test</env>
      <url>192.13.24.200</url>
      <port>3306</port>
      <login>admin</login>
      <password>password</password>
    </database>
  </database>
</config>
```

L'index du premier élément vaut 0. L'index à utiliser doit être indiqué entre parenthèses.

Exemple :

```
XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
String url = config.getString("databases.database(1).url");

String port = config.getString("databases.database(1).port");
```

La méthode getMaxIndex(), héritée de la classe org.apache.commons.HierarchicalConfiguration, qui attend en paramètre le nom d'une clé renvoie le nombre d'éléments présents dans le fichier de configuration.

Il est possible de récupérer la valeur à partir d'un attribut d'un tag en utilisant une syntaxe particulière dans laquelle le nom de l'attribut précédé d'un arobase est indiqué entre crochets.

Exemple :

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <config>
3.     <database url="127.0.0.1" port="3306" login="admin" password="password"/>
4. </config>
```

Exemple :

```
1. XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
2. String url = config.getString("databases.database.[@url]");
3. String port = config.getString("databases.database.[@port]");
```

La notation par point pour accéder à un élément de la configuration fonctionne bien pour des cas simples mais elle s'avère limitative pour des cas plus complexes. Pour ces cas, il est possible d'utiliser XPath en associant à la configuration une instance de type XPathExpressionEngine.

Exemple :

```
1. XMLConfiguration config = new XMLConfiguration("dbConfig.xml");
2. config.setExpressionEngine(new XPathExpressionEngine());
3. config.getString("databases/database[env = 'test']/url");
4. config.getString("databases/database[env
5. = 'test']/port");
```

Les commentaires du fichier XML d'origine qui a été utilisé pour charger la configuration sont préservés lors de la sauvegarde d'une configuration modifiée.

99.1.2.2. Les configurations dans la base de données

La classe org.apache.commons.configuration.DatabaseConfiguration encapsule une configuration stockée dans une base de données.

La table qui va stocker des données de la configuration doit contenir deux colonnes : une pour la clé et une autre pour la valeur. Une troisième colonne optionnelle peut être utilisée pour contenir le nom de la configuration : ceci permet de stocker plusieurs configurations dans la table.

Plusieurs constructeurs de la classe DatabaseConfiguration permettent de fournir les informations utiles pour accéder aux informations dans la table :

- DataSource datasource : connexion à la base de données
- String table : nom de la table
- String nameColumn : nom de la colonne qui contient le nom de la configuration
- String keyColumn : nom de la colonne qui contient la clé
- String valueColumn : nom de la colonne qui contient la valeur
- String name : le nom de la configuration

Un accès à la base de données est réalisé à chaque demande de la valeur d'une clé, ce qui peut engendrer de nombreux accès à la base

99.1.2.3. Les configurations dans une instance de type Map

La classe org.apache.commons.configuration.MapConfiguration encapsule une configuration dont la source est une collection existante de type Map.

Les données contenues dans la Map doivent respecter certaines conventions pour être utilisables comme source de la configuration, notamment que la clé soit de type String et que les valeurs ne soient pas de type List.

La classe MapConfiguration possède deux constructeurs : un premier qui attend en paramètre un objet de type Map<String, Object> et un second qui attend en paramètre un objet de type Properties.

Les clés de la configuration sont directement mappées sur les clés de la map : par exemple, la méthode getProperty() invoque directement la méthode get() sur l'instance de la Map.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
```

```
02.
03. import java.util.HashMap;
04. import java.util.Map;
05.
06. import org.apache.commons.configuration.MapConfiguration;
07.
08. public class TestMapConfiguration {
09.
10.   public static void main(final String[] args) {
11.     final Map<String, Object> donnees = new HashMap<String, Object>();
12.     donnees.put("valeurs", new Object[] { "valeur1", "valeur2", "valeur3" });
13.     donnees.put("maCle", "maValeur");
14.
15.     final MapConfiguration config = new MapConfiguration(donnees);
16.     System.out.println("valeurs=" + config.getList("valeurs"));
17.     System.out.println("maCle=" + config.getString("maCle"));
18.   }
19. }
```

Comme l'instance de type Map fournie au constructeur est utilisée pour stocker les valeurs, c'est le type de cette instance qui détermine si les opérations sont thread-safe ou non.

99.1.2.4. JNDI

La classe org.apache.commons.configuration.JNDIConfiguration utilise les données d'un service de nommage accessible grâce à JNDI comme source pour la configuration.

La propriété context permet de préciser le Context JNDI pour accéder à la ressource.

La propriété prefix permet de préciser le préfixe à utiliser dans le Context lors des recherches.

La classe JNDIConfiguration possède quatre constructeurs : le constructeur par défaut et trois surcharges qui permettent de préciser le Context et/ou le préfixe.

Les valeurs sont recherchées dans la Context JNDI en utilisant le préfixe et la valeur de la clé.

La configuration encapsulée dans un JNDIConfiguration est en lecture seule. Les méthodes addPropertyDirect() et setProperty() lèvent une exception de type UnsupportedOperationException.

La méthode getBaseContext() renvoie le Context JNDI pour lequel le préfixe est appliqué.

99.1.2.5. Les variables systèmes

La classe org.apache.commons.configuration.SystemConfiguration utilise comme source de la configuration les propriétés système et les propriétés de la JVM fournies avec l'option -Dcle=valeur.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.EnvironmentConfiguration;
04.
05. public class TestEnvironmentConfiguration {
06.   public static void main(final String[] args) {
07.     final EnvironmentConfiguration config = new EnvironmentConfiguration();
08.     System.out.println("java home=" + config.getString("JAVA_HOME"));
09.     System.out.println("os=" + config.getString("OS"));
10.   }
11. }
```

99.1.3. Les beans dynamiques

La classe org.apache.commons.configuration.beanutils.ConfigurationDynaBean permet d'encapsuler une configuration et de l'utiliser comme un DynaBean. Un DynaBean permet d'obtenir et de modifier les propriétés d'un bean de manière dynamique.

Pour créer une instance de type ConfigurationDynaBean, il suffit d'invoquer son constructeur en lui passant en paramètre la configuration.

Exemple :

```java
package com.jmdoudoux.test.commons.configuration;

import java.util.HashMap;
import java.util.Map;

import org.apache.commons.configuration.MapConfiguration;
import org.apache.commons.configuration.beanutils.ConfigurationDynaBean;

public class TestConfigurationDynaBean {

  public static void main(final String[] args) {
    final Map<String, Object> donnees = new HashMap<String, Object>();
    donnees.put("valeurs", new Object[] { "valeur1", "valeur2", "valeur3" });

    final MapConfiguration config = new MapConfiguration(donnees);

    final ConfigurationDynaBean bean = new ConfigurationDynaBean(config);
    final Object value = bean.get("valeurs", 1);
    System.out.println(value);
  }
}
```

99.1.4. Les configurations composites

Il est fréquent que la configuration provienne de plusieurs sources : la classe org.apache.commons.configuration.CompositeConfiguration permet d'agréger plusieurs sources pour obtenir une configuration avec un seul point d'entrée.

La contrepartie est qu'une fois la configuration chargée, il n'est plus possible d'utiliser les spécificités de chacune des configurations qui sont agrégées : seules les fonctionnalités communes définies par l'interface Configuration sont utilisables.

La méthode addConfiguration() permet d'ajouter une configuration à la composition.

Exemple :

```java
CompositeConfiguration config = new CompositeConfiguration();
config.addConfiguration(new SystemConfiguration());
config.addConfiguration(new PropertiesConfiguration("monApp.properties"));
```

Une propriété demandée à une composition est recherchée successivement dans les configurations dans leur ordre d'ajout. Dès que la propriété est trouvée dans une configuration, sa valeur est retournée. Ce mode de fonctionnement permet de mettre en place un mécanisme de redéfinition de valeurs pour une même propriété.

Exemple : le fichier maConfig.properties

```
ma.valeur=100
ma.valeur2=200
```

Exemple : le fichier maConfigDefaut.properties

```
ma.valeur=10
ma.valeur2=20
```

Exemple :

```java
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.CompositeConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.PropertiesConfiguration;

public class TestCompositeConfiguration {

  public static void main(final String[] args) {
    final CompositeConfiguration config = new CompositeConfiguration();
    try {
      config.addConfiguration(new PropertiesConfiguration("maConfig.properties"));
      config.addConfiguration(new PropertiesConfiguration("maConfigDefaut.properties"));

      System.out.println("ma.valeur=" + config.getString("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
ma.valeur=100
```

Une CompositeConfiguration est donc particulièrement utile pour gérer une hiérarchie de configuration. Comme une propriété est recherchée dans l'ordre dans lequel les configurations ont été ajoutées à la composition, il suffit que la dernière composition contienne les valeurs par défaut. Si la propriété n'est pas redéfinie dans les autres configurations, c'est celle obtenue de la configuration par défaut qui sera retournée.

Exemple : le fichier maConfig.properties

```
ma.valeur2=200
```

Exemple : le fichier maConfigDefaut.properties

```
ma.valeur=10
ma.valeur2=20
```

Résultat :

```
ma.valeur=10
```

Si une propriété de la composition doit pouvoir être modifiée et sauvegardée, il est nécessaire de fournir une configuration en paramètre du constructeur utilisé pour créer l'instance de type CompositionConfiguration.

Toutes les modifications faites dans la configuration sont réalisées dans la source précisée en paramètre du constructeur (in memory configuration). La méthode getInMemoryConfiguration() pour d'obtenir l'instance qui encapsule cette source.

99.1.5. Les configurations hiérarchiques

Plusieurs sources de configuration utilisent une structure hiérarchique ou arborescente pour organiser leurs données, par exemple celles utilisant XML. Ce type de configuration est encapsulé dans une classe qui hérite de la classe HierarchicalConfiguration.

Plusieurs configurations sont de type hiérarchique :

- AbstractHierarchicalFileConfiguration : HierarchicalINIConfiguration, MultiFileHierarchicalConfiguration, PatternSubtreeConfigurationWrapper, PropertyListConfiguration, XMLConfiguration, XMLPropertyListConfiguration
- CombinedConfiguration
- SubnodeConfiguration

Les éléments imbriqués sont désignés en utilisant la notation par point en ignorant l'élément racine.

Pour accéder à un attribut d'un tag XML, il faut utiliser une syntaxe proche de XPath :

- cle(index) : permet d'accéder à l'élément précisé par index. Le premier élément possède l'index 0
- [@nom_attribut] : permet d'accéder à la valeur de l'attribut dont le nom est nom_attribut

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import java.util.List;
04.
05. import org.apache.commons.configuration.ConfigurationException;
06. import org.apache.commons.configuration.HierarchicalConfiguration;
07. import org.apache.commons.configuration.XMLConfiguration;
08. import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;
09.
10. public class TestXMLConfiguration {
11.
12.   public static void main(final String[] args) {
13.     try {
14.       final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
15.      System.out.println(config.getList("environnements.environnement(0).champs.champ"));
16.       System.out.println(config.getList("paire"));
17.       System.out.println(config.getString("environnements.environnement(1)[@nom]"));
18.
19.     } catch (final ConfigurationException e) {
20.       e.printStackTrace();
21.     }
22.   }
```

```
23. }
```

Résultat :

```
1. [local1, local2, local3]
2. [2, 4, 6, 8]
3. test
```

Parfois le nom d'une clé peut être long si l'arborescence de la configuration est complexe. Pour faciliter l'utilisation d'une portion de la configuration, il est possible d'utiliser la méthode configurationAt() . Elle renvoie une instance de type HierarchicalConfiguration qui encapsule la portion de la configuration sous la clé fournie en paramètre.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import java.util.List;
04.
05. import org.apache.commons.configuration.ConfigurationException;
06. import org.apache.commons.configuration.HierarchicalConfiguration;
07. import org.apache.commons.configuration.XMLConfiguration;
08. import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;
09.
10. public class TestXMLConfiguration {
11.
12.   public static void main(final String[] args) {
13.     try {
14.       final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
15.
16.       final HierarchicalConfiguration sub =
17.         config.configurationAt("environnements.environnement(1).champs");
18.       final List<Object> champs = sub.getList("champ");
19.       System.out.println(champs);
20.
21.     } catch (final ConfigurationException e) {
22.       e.printStackTrace();
23.     }
24.   }
25. }
```

Résultat :

```
1. [test1, test2, test3]
```

Cette syntaxe est utilisable aussi pour modifier la configuration. Pour ajouter un nouvel élément dans une liste en première position, il faut utiliser un index avec la valeur -1.

Il est nécessaire d'échapper certains caractères dans le nom des clés ou dans les valeurs.

Par défaut, le caractère point est utilisé comme séparateur entre les différents éléments qui composent la clé. Cependant le caractère point peut être utilisé dans le nom d'un tag XML. Dans ce cas, une exception de type NoSuchElementException est levée.

Exemple :

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <config>
3.   <ma.valeur>100</ma.valeur>
4. </config>
```

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.ConfigurationException;
04. import org.apache.commons.configuration.XMLConfiguration;
05. import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;
06.
07. public class TestXMLConfiguration {
08.
09.   public static void main(final String[] args) {
10.     try {
11.       final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
12.       System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
13.     } catch (final ConfigurationException e) {
14.       e.printStackTrace();
15.     }
16.   }
17. }
```

Résultat :

```
Exception in thread "main" java.util.NoSuchElementException: 'ma.valeur' doesn't map
to an existing object
        at org.apache.commons.configuration.AbstractConfiguration.getLong(Abstract
Configuration.java:862)
        at com.jmdoudoux.test.commons.configuration.TestXMLConfiguration.main(Test
XMLConfiguration.java:16)
```

Pour changer ce comportement, il faut doubler le caractère point dans le nom de la clé pour déspécialiser le caractère point.

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("ma.valeur=" + config.getLong("ma..valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Par défaut, la virgule est utilisée comme séparateur pour une liste de valeurs dans le corps d'un tag.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
  <mavaleur>100,200</mavaleur>
</config>
```

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("mavaleur=" + config.getLong("mavaleur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
Exception in thread "main" java.util.NoSuchElementException: 'ma.valeur' doesn't map
to an existing object
        at org.apache.commons.configuration.AbstractConfiguration.getLong(Abstract
Configuration.java:862)
        at com.jmdoudoux.test.commons.configuration.TestXMLConfiguration.main(Test
XMLConfiguration.java:16)
```

Par défaut, le caractère virgule est utiliser comme séparateur de plusieurs éléments d'une valeur.

Exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<config>
  <mavaleur>100,123456789</mavaleur>
</config>
```

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
      System.out.println("mavaleur=" + config.getString("mavaleur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
mavaleur=100
```

Si les clés ne doivent pas être considérées comme pouvant avoir plusieurs valeurs, il faut désactiver la fonctionnalité en invoquant les méthodes setDelimiterParsingDisabled() et setAttributeSplittingDisabled() avec la valeur true en paramètre.

La désactivation de cette fonctionnalité doit impérativement se faire avant le chargement du fichier XML.

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();

      config.setDelimiterParsingDisabled(true);
      config.setAttributeSplittingDisabled(true);
      config.load("maConfig.xml");

      System.out.println("mavaleur=" + config.getString("mavaleur"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
mavaleur=100,123456789
```

Pour les clés qui doivent avoir plusieurs valeurs, il faut alors utiliser un tag pour chaque valeur dans le fichier de configuration XML.

Pour des besoins plus complexes, il est possible d'utiliser une instance de type ExpressionEngine. Un objet de type ExpressionEngine est responsable de l'interprétation des clés dans une source de la configuration.

La méthode setExpressionEngine() permet de préciser une instance de type ExpressionEngine à utiliser. Les clés utilisées doivent alors respecter la syntaxe exploitable par l'ExpressionEngine.

La syntaxe utilisée par défaut est définie dans une instance de type DefaultExpressionEngine.

La classe DefaultExpressionEngine possède plusieurs attributs :

- AttributStart : le marqueur de début d'attribut
- AttributEnd : le marqueur de fin d'attribut
- IndexStart : le marqueur de début d'index

- IndexEnd : le marqueur de fin d'index
- PropertyDelimiter : le délimiteur de propriétés

La méthode setDefaultExpressionEngine() de la classe HierarchicalConfiguration permet de modifier l'ExpressionEngine utilisé par défaut.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.ConfigurationException;
04. import org.apache.commons.configuration.HierarchicalConfiguration;
05. import org.apache.commons.configuration.XMLConfiguration;
06. import org.apache.commons.configuration.tree.DefaultExpressionEngine;
07.
08. public class TestDefaultExpressionEngine {
09.
10.   public static void main(final String[] args) {
11.     final DefaultExpressionEngine engine = new DefaultExpressionEngine();
12.
13.     engine.setPropertyDelimiter("/");
14.     engine.setIndexStart("{");
15.     engine.setIndexEnd("}");
16.     engine.setAttributeStart("@");
17.     engine.setAttributeEnd(null);
18.
19.     HierarchicalConfiguration.setDefaultExpressionEngine(engine);
20.
21.     final XMLConfiguration config = new XMLConfiguration();
22.     try {
23.       config.load("maConfig.xml");
24.       System.out.println(config.getString("environnements/environnement{0}/champs/champ"));
25.       System.out.println(config.getString("environnements/environnement{0}@name"));
26.     } catch (final ConfigurationException e) {
27.       e.printStackTrace();
28.     }
29.   }
30. }
```

Résultat :

```
1. local1
2. null
```

La classe XPathExpressionEngine encapsule un ExpressionEngine qui exploite la syntaxe XPath.

Son utilisation requiert que la bibliothèque Commons JXPath soit ajoutée au classpath.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.ConfigurationException;
04. import org.apache.commons.configuration.XMLConfiguration;
05. import org.apache.commons.configuration.tree.xpath.XPathExpressionEngine;
06.
07. public class TestXPathExpressionEngine {
08.
09.   public static void main(final String[] args) {
10.     try {
11.       final XMLConfiguration config = new XMLConfiguration();
12.       config.setExpressionEngine(new XPathExpressionEngine());
13.       config.load("maConfig.xml");
14.
15.       System.out.println(config.getList(
16.         "environnements/environnement[@nom='test']/champs/champ"));
17.
18.     } catch (final ConfigurationException e) {
19.       e.printStackTrace();
20.     }
21.   }
22. }
```

Résultat :

```
1. [test1, test2, test3]
```

Il est possible de valider le fichier XML via sa DTD en passant la valeur true à la méthode setValidating().

Exemple :

```java
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();
      config.setValidating(true);
      config.load("maConfig.xml");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Il est possible de valider le fichier XML via son schéma en passant la valeur true à la méthode setSchemaValidating().

Exemple :

```java
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.reloading.FileChangedReloadingStrategy;

public class TestXMLConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration config = new XMLConfiguration();
      config.setSchemaValidating(true);
      config.load("maConfig.xml");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

99.1.6. La classe CombinedConfiguration

La classe CombinedConfiguration permet de combiner plusieurs sources de configuration. Elle est similaire à la classe CompositeConfiguration avec plusieurs différences :

- c'est une configuration hiérarchique (HierarchicalConfiguration)
- la façon dont la combinaison est faite est configurable en utilisant une instance de type NodeCombiner
- un nom peut être assigné à chaque source pour permettre leur accès
- chaque source peut avoir un préfixe qui sera ajouté à chacune de leur propriété

La classe CombinedConfiguration propose une vue logique des propriétés des sources de configuration qui lui sont associées. Cette vue est réalisée grâce à l'utilisation d'une instance de type NodeCombiner.

Si une des configurations source est modifiée alors la vue est invalidée et reconstruite par le NodeCombiner si nécessaire.

Plusieurs classes filles de type NodeCombiner sont proposées en standard :

- OverrideCombiner
- MergeCombiner
- UnionCombiner

La méthode addConfiguration() permet d'ajouter une Configuration à la combinaison. Une surcharge attend aussi en paramètre un nom. Ce nom peut être utilisé en paramètre de la méthode getConfiguration() pour obtenir la configuration correspondante.

La méthode addListNode() permet de préciser qu'un élément doit être considéré comme une liste.

Les exemples de cette section vont utiliser deux fichiers de configurations qui seront combinés.

Exemple : Le fichier configUsers.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <database>
    <tables>
      <table>
        <name>users</name>
        <fields>
```

```
            <field>
              <name>user_id</name>
            </field>
            <field>
              <name>user_name</name>
            </field>
            </fields>
        </table>
      </tables>
    </database>
</configuration>
```

Exemple : le fichier configGroups.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <database>
    <tables>
      <table>
        <name>groups</name>
        <fields>
          <field>
            <name>group_id</name>
          </field>
          <field>
            <name>user_id</name>
          </field>
        </fields>
      </table>
    </tables>
  </database>
</configuration>
```

La classe MergeCombiner implémente une combinaison par fusion.

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.CombinedConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.MergeCombiner;
import org.apache.commons.configuration.tree.NodeCombiner;

public class TestCombinedConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration confUsers = new XMLConfiguration("configUsers.xml");
      final XMLConfiguration confGroups = new XMLConfiguration("configGroups.xml");

      final NodeCombiner combiner = new MergeCombiner();

      final CombinedConfiguration cc = new CombinedConfiguration(combiner);
      cc.addConfiguration(confUsers, "users");
      cc.addConfiguration(confGroups);

      System.out.println(cc.getList("database.tables.table.name"));
      System.out.println(cc.getList("database.tables.table.fields.field.name"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[users]
[user_id, user_name]
```

La classe UnionCombiner implémente une combinaison par union. Les éléments des différentes sources sont ajoutés dans la vue.

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.CombinedConfiguration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.XMLConfiguration;
import org.apache.commons.configuration.tree.NodeCombiner;
```

```java
import org.apache.commons.configuration.tree.UnionCombiner;

public class TestCombinedConfiguration {

  public static void main(final String[] args) {
    try {
      final XMLConfiguration confUsers = new XMLConfiguration("configUsers.xml");
      final XMLConfiguration confGroups = new XMLConfiguration("configGroups.xml");

      final NodeCombiner combiner = new UnionCombiner();

      final CombinedConfiguration cc = new CombinedConfiguration(combiner);
      cc.addConfiguration(confUsers, "users");
      cc.addConfiguration(confGroups);

      System.out.println(cc.getList("database.tables.table.name"));
      System.out.println(cc.getList("database.tables.table.fields.field.name"));

    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
[users, groups]
[user_id, user_name, group_id, user_id]
```

La classe OverrideCombiner implémente une combinaison par surcharge : les éléments de la première configuration sont prépondérants sur ceux des configurations suivantes. Un élément de la seconde configuration ne peut être inclus dans la combinaison que s'il n'existe pas dans la première configuration.

Il n'est pas recommandé de faire des modifications directement dans une instance de type CombinedConfiguration même si rien ne l'empêche. Du fait de l'utilisation du NodeCombiner le résultat peut être inattendu : il est donc préférable de faire les modifications dans les configurations qui sont contenues dans la CombinedConfiguration.

99.1.7. Les configurations dynamiques

Les configurations dynamiques permettent de créer une instance de type CompositeConfiguration dont les différentes sources sont précisées dans un fichier XML.

Exemple :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- config.xml -->
<configuration>

  <env />
  <xml fileName="config.xml" />
  <properties fileName="config.properties" />
</configuration>
```

Le tag racine est le tag <configuration>.

Chacune des sources qui doivent composer la configuration est précisée grâce à un tag dédié :

- <env> :
- <xml> : La propriété fileName permet de préciser le fichier XML
- <properties> : La propriété fileName permet de préciser le fichier .properties

Ces tags peuvent être directement des fils du tag racine ou de fils d'un tag <override>.

Exemple :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- config.xml -->
<configuration>
  <override>
    <env />
    <xml fileName="config.xml" />
    <properties fileName="config.properties"/>
  </override>
</configuration>
```

Pour utiliser cette fonctionnalité, il est nécessaire d'ajouter la bibliothèque Commons Digester dans le classpath.

Le format général du fichier de définition de la configuration est de la forme :

Exemple :

```
1. <?xml version="1.0" encoding="ISO-8859-1" ?>
2. <configuration>
3.   <header> ...</header>
4.   <override> ... </override>
5.   <additional> ... </additional>
6. </configuration>
```

Tous les tags fils <header>, <override> et <additional> sont optionnels.

La déclaration des différences sources de configuration à utiliser se fait en utilisant des tags dédiés par type de source :

- properties : ajouter un fichier .properties dont le nom est précisé grâce au tag fileName. Si le nom du fichier se termine par .xml alors c'est une instance de type XMLPropertiesConfiguration qui est utilisée sinon c'est une instance de type PropertiesConfiguration
- xml : ajouter un fichier .xml dont le nom est précisé grâce au tag fileName. C'est une instance de type XMLConfiguration qui est utilisée
- jndi : ajouter une ressource JNDI. L'attribut prefix permet de préciser la racine dans l'arborescence JNDI. C'est une instance de type JNDIConfiguration qui est utilisée
- plist : ajouter un fichier de type .plist dont le nom est précisé grâce au tag fileName. Si le fichier est au format xml alors c'est une instance de type XMLPropertyListConfiguration qui est utilisée sinon c'est une instance de type PropertyListConfiguration
- system : ajouter un accès aux propriétés systèmes. C'est une instance de type SystemConfiguration qui est utilisée
- ini : ajouter un fichier de type .ini. C'est une instance de type HierarchicalINIConfiguration qui est utilisée
- env : ajouter un accès aux propriétés d'environnement

Pour chaque source, il est possible de modifier des propriétés

Exemple :

```
01. <?xml version="1.0" encoding="ISO-8859-1" ?>
02. <configuration>
03.   <system />
04.   <properties fileName="maConfig.properties" throwExceptionOnMissing="true">
05.     <reloadingStrategy refreshDelay="60000" config-class=
06.       "org.apache.commons.configuration.reloading.FileChangedReloadingStrategy" />
07.   </properties>
08.   <properties fileName="maConfigDefaut.properties" />
09. </configuration>
```

L'attribut config-name permet de définir un nom pour la configuration.

Il est possible d'effectuer un remplacement dynamique de valeurs avec celles de variables système en précisant le nom de la variable entre ${ et }. Dans ce cas, il est nécessaire que ces variables soient chargées en utilisant une source <system>.

Exemple :

```
1. <?xml version="1.0" encoding="ISO-8859-1" ?>
2. <configuration>
3.   <system/>
4.   <properties fileName="${CONFIG_FILE}"/>
5. </configuration>
```

La recherche d'une propriété se fait dans l'ordre de déclaration des sources dans le fichier XML.

Par défaut, si une source définie dans le fichier de configuration n'est pas trouvée lors du chargement, alors une exception de type ConfigurationException est levée. Il est possible de définir une source optionnelle en utilisant l'attribut config-optional avec la valeur true.

Exemple :

```
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <configuration>
3.   <properties fileName="user.properties" config-optional="true"/>
4.   <properties fileName="default.properties"/>
5. </configuration>
```

Si la source est configurée comme optionnelle et qu'elle n'est pas trouvée à son chargement alors aucune exception n'est levée est un message de type warning est inséré dans le log.

Il est possible d'unir plusieurs sources de configuration comme si ce n'était qu'une seule source. Les différentes sources doivent être définies en tant que tag fils du tag <additional>. L'attribut config-at permet de préciser avec quelle source elle doit être fusionnée.

Par défaut, le tag <override> peut être omis sauf si un tag <additional> est présent.

99.1.7.1. La classe ConfigurationFactory

La classe org.apache.commons.configuration.ConfigurationFactory permet de construire des configurations dynamiques.

Exemple : le fichier config.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<configuration>
  <system/>
  <properties fileName="maConfig.properties"/>
  <properties fileName="maConfigDefaut.properties"/>
</configuration>
```

Exemple :

```
package com.jmdoudoux.test.commons.configuration;

import org.apache.commons.configuration.Configuration;
import org.apache.commons.configuration.ConfigurationException;
import org.apache.commons.configuration.ConfigurationFactory;

public class TestConfigurationFactory {

  public static void main(final String[] args) {

    final ConfigurationFactory factory = new ConfigurationFactory("config.xml");
    try {
      final Configuration config = factory.getConfiguration();

      System.out.println("ma.valeur=" + config.getString("ma.valeur"));
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }
}
```

A partir de la version 1.3, la classe ConfigurationFactory est deprecated : il faut utiliser la classe DefaultConfigurationBuilder.

99.1.7.2. La classe DefaultConfigurationBuilder

La classe ConfigurationBuilder permet de créer une configuration à partir des informations fournies dans un fichier XML.

Le plus simple est de créer une instance de type DefaultConfigurationBuilder. Cette classe propose plusieurs constructeurs dont certains permettent de préciser le fichier de configuration.

La méthode setFile() permet de préciser le fichier de configuration si celui-ci n'a pas été fourni en paramètre du constructeur.

La méthode getConfiguration() permet de charger la configuration en créant une instance de type Configuration.

Une surcharge de la méthode getConfiguration(), qui attend en paramètre un booléen précisant le fichier XML devant être lu, renvoie une instance de type CombinedConfiguration.

Exemple :

```
DefaultConfigurationBuilder builder = new DefaultConfigurationBuilder("config.xml");
CombinedConfiguration config = builder.getConfiguration(true);
```

99.1.8. Les sous-ensembles d'une configuration

Il est possible d'utiliser un sous-ensemble d'une configuration qui correspond aux éléments qui commencent par un même préfixe.

Il y a deux façons d'obtenir un sous-ensemble :

- en créant une instance de type SubsetConfiguration dont les constructeurs attendent en paramètre la configuration source et le préfixe
- en invoquant la méthode subset() qui attend en paramètre le préfixe et renvoie une instance de type SubsetConfiguration

Une instance de type Subsetconfiguration implémente l'interface Configuration. Il faut retirer le préfixe qui est contenu dans la configuration originale pour accéder aux éléments correspondants dans le sous-ensemble.

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.Configuration;
04. import org.apache.commons.configuration.ConfigurationException;
05. import org.apache.commons.configuration.XMLConfiguration;
06.
07. public class TestSubsetConfiguration {
08.
09.   public static void main(final String[] args) {
10.     try {
11.       final XMLConfiguration config = new XMLConfiguration("maConfig.xml");
12.       String url = config.getString("data.database.url");
13.       System.out.println("url = " + url);
14.
15.       final Configuration dbConfig = config.subset("data.database");
16.       url = dbConfig.getString("url");
17.       System.out.println("url = " + url);
18.
19.     } catch (final ConfigurationException e) {
20.       e.printStackTrace();
21.     }
22.   }
23. }
```

Résultat :

```
1. url = 127.0.0.1
2. url = 127.0.0.1
```

99.1.9. Les événements sur la configuration

La classe AbstractConfiguration propose des fonctionnalités pour enregistrer des listeners sur des événements qui permettent d'être notifié lors d'un changement sur des données de la configuration.

L'interface ConfigurationListener définit la méthode configurationChanged() qui sera invoquée si le listener est enregistré et qu'un événement de modification de la configuration est émis. La méthode configurationChanged() possède un paramètre de type ConfigurationEvent qui encapsule des informations sur la modification :

- source object : généralement la configuration qui a été modifiée
- event type : constante numérique qui précise le type d'événement (EVENT_ADD_PROPERTY, EVENT_SET_PROPERTY, EVENT_CLEAR_PROPERTY, EVENT_CLEAR définies dans la classe AbstractConfiguration, EVENT_RELOAD définie dans la classe AbstractFileConfiguration)
- le nom de la propriété concernée si elle est connue
- la valeur de la propriété si elle est connue

La méthode isBeforeUpdate() renvoie un booléen qui précise si l'événement est émis avant une modification (valeur true) ou après (valeur false).

Exemple :

```
01. package com.jmdoudoux.test.commons.configuration;
02.
03. import org.apache.commons.configuration.ConfigurationException;
04. import org.apache.commons.configuration.PropertiesConfiguration;
05. import org.apache.commons.configuration.event.ConfigurationEvent;
06. import org.apache.commons.configuration.event.ConfigurationListener;
07.
08. public class TestConfigurationListener {
09.
10.   public static void main(final String[] args) {
11.     try {
12.       final PropertiesConfiguration config =
13.           new PropertiesConfiguration("maConfig.properties");
14.       config.addConfigurationListener(new ConfigurationListener() {
15.
16.         @Override
17.         public void configurationChanged(final ConfigurationEvent event) {
18.
19.           if (!event.isBeforeUpdate()) {
20.             System.out.println("Event: ");
21.             System.out.println("  Type = " + event.getType());
22.             System.out.println("  Source = " + event.getSource());
23.             if (event.getPropertyName() != null) {
24.               System.out.println("  Property name = " + event.getPropertyName());
25.             }
26.             if (event.getPropertyValue() != null) {
27.               System.out.println("  Property value = " + event.getPropertyValue());
```

```
            }
            System.out.println("");
          }
        }
      });

      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.setProperty("ma.valeur", 100);
      System.out.println("ma.valeur=" + config.getLong("ma.valeur"));
      config.addProperty("nouvelle.valeur", "200");
    } catch (final ConfigurationException e) {
      e.printStackTrace();
    }
  }

}
```

Résultat :

```
ma.valeur=10
Event:
  Type = 3
  Source = org.apache.commons.configuration.PropertiesConfiguration@ca2dce
  Property name = ma.valeur
  Property value = 100

ma.valeur=100
Event:
  Type = 1
  Source = org.apache.commons.configuration.PropertiesConfiguration@ca2dce
  Property name = nouvelle.valeur
  Property value = 200
```

L'interface ConfigurationErrorListener définit la méthode configurationError() qui sera invoquée si le listener est enregistré et qu'une exception est levée par un sous système utilisé par la configuration. La méthode configurationError() possède un paramètre de type ConfigurationErrorEvent qui encapsule l'exception. Celle-ci peut être obtenue en invoquant la méthode getCause() qui renvoie une instance de Throwable.

99.1.10. La classe ConfigurationUtils

La classe org.apache.commons.configuration.ConfigurationUtils est une classe proposant des utilitaires dont toutes les méthodes sont statiques.

Plusieurs méthodes concernent les configurations.

Méthode	Rôle
void append(Configuration source, Configuration cible)	Ajouter la configuration source dans la configuration cible. Les valeurs des clés existantes dans la configuration cible sont remplacées par les valeurs correspondantes dans la configuration source
Configuration cloneConfiguration(Configuration config)	Retourner un objet clone de la configuration
HierarchicalConfiguration convertToHierarchical(Configuration conf)	Convertir une configuration en configuration hiérarchique
HierarchicalConfiguration convertToHierarchical(Configuration conf, ExpressionEngine engine)	Convertir une configuration en configuration hiérarchique en utilisant l'ExpressionEngine fournie en paramètre
copy(Configuration source, Configuration cible)	Copier la configuration source dans la configuration cible
dump(Configuration conf, PrintStream out) dump(Configuration conf, PrintWriter out)	Envoyer un dump de la configuration, sous la forme clé=valeur dans le flux fourni en paramètre
void enableRuntimeExceptions(Configuration src)	Activer les exceptions de type Runtime pour la configuration
toString(Configuration conf)	Renvoyer un dump de la configuration, sous la forme clé=valeur

Plusieurs méthodes concernent les fichiers : getFile(), getURL(), locate(), fileFromURL().

99.2. Apache Commons CLI

La bibliothèque Commons CLI permet de facilement interpréter les différents arguments qui dont passés via la ligne de commande à une application standalone.

Ces arguments peuvent permettre de définir des options, fournir des valeurs, ... Leur exploitation n'est toujours facile car elles peuvent avoir un court et/ou long, une ou plusieurs valeurs, être obligatoire ou non, ...

La bibliothèque Commons CLI (Command Line Interface) permet d'analyser et d'obtenir les options passées au programme par la ligne de commande

Le site du projet est à l'url http://commons.apache.org/cli/

Commons CLI propose le support de plusieurs formats pour les options :

- style POSIX : une option est identifiée par une lettre. Chaque option peut être précédée d'un caractère tiret ou les options peuvent être concaténées devant un seul caractère tiret

(exemple : tar -xvf monarchive.tar)

- style GNU : le format long est précédé d'un double tiret

(exemple : tar --verbose --extract --file=monarchive.tar)

- Propriété Java

(exemple : -Dpropriete=valeur)

Les arguments passés à une ligne de commande peuvent être de deux natures :

- options ou flags : généralement préfixé par un caractère tiret ou deux caractères tirets (sous Unix) ou un caractère slash sous Windows
- valeurs d'un argument : les différentes valeurs sont concaténées avec un caractère de séparation, généralement un caractère virgule

Une option et sa valeur peuvent être concaténées avec un caractère de séparation, généralement un caractère égal.

Une option peut être associée à une ou plusieurs valeurs.

Les traitements relatifs aux options de la ligne de commandes avec CLI se font en trois étapes :

- définition des options
- analyse des options fournies en paramètres du programme
- obtention des options et de leurs valeurs

Commons CLI permet aussi d'afficher un résumé des options, ce qui est particulièrement utile notamment si l'analyse des options échoue pour fournir une aide sur les options utilisables.

CLI ne permet pas de gérer certains cas complexes ou particuliers comme par exemple :

- le fait que la valeur est concaténée à l'option (exemple : java -Xmx128m
- s'assurer de la cohérence de la présence d'autres options selon qu'un option particulière soit présente
- vérifier et convertir au besoin la ou les valeurs associées à une option

La version utilisée dans cette section est la 1.2 : elle peut être téléchargée à l'url http://commons.apache.org/cli/download_cli.cgi

99.2.1. La définition des options

La classe Options est un conteneur qui va encapsuler les différentes instances de type org.apache.commons.cli.Option.

Pour créer une nouvelle instance de type org.apache.commons.cli.Options, il suffit d'invoquer son constructeurs sans arguments.

Une option est encapsulée dans une instance type Option qui possède plusieurs propriétés :

Propriété	Rôle
String opt	Nom court de l'option
String longOpt	Nom long de l'option
String description	Description de l'option
Boolean required	Définir si l'option est obligatoire
Boolean arg	Définir si l'option possède un argument obligatoire
Boolean args	Définir si l'option peut avoir plusieurs arguments
Boolean optionalArg	Définir si l'option possède un argument optionnel
String argName	Nom de l'argument
Char valueSeparatorChar	Définir le caractère de séparation des différentes valeurs de l'argument
Object type	Le type de la valeur de l'argument
String value	La valeur de l'option
String[] values	Les valeurs de l'option

La méthode addOption() de la classe Options permet d'ajouter la définition d'une nouvelle option. Elle possède plusieurs surcharges.

Une option est encapsulée dans la classe Option.

Une instance de type Option peut être créée de trois manières pour être ajoutée dans une instance de type Options :

- invoquer un de ses constructeurs
- utiliser une des surcharges de la méthodes addOption() de la classe Options qui attend en paramètres les principales caractéristiques de l'option
- utiliser la classe OptionBuilder qui implémente le motif de conception

La classe Option possède plusieurs constructeurs :

Constructeur	Rôle
Option(String opt, boolean hasArg, String description)	Créer une instance de type Option avec les paramètres fournis : • opt : le nom de l'option tel qu'il sera fourni par l'utilisateur • hasArg : l'option possède un argument ou non • description : la description de l'option qui sera utilisée pour afficher une aide à l'utilisateur • longOpt :
Option(String opt, String description)	
Option(String opt, String longOpt, boolean hasArg, String description)	

La méthode addOption() possède une surcharge qui attend en paramètre une instance de type Option.

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import org.apache.commons.cli.Option;
04. import org.apache.commons.cli.Options;
05.
06. public class App {
07.
08.   public static void main(final String[] args) {
09.     final Options options = new Options();
10.     final Option optionNom = new Option("n", true, "[nom] votre nom");
11.     options.addOption(optionNom);
12.   }
13. }
```

La méthode addOption() de la classe Options possède deux surcharges qui attendent plusieurs paramètres qui sont le principales caractéristiques de l'option.

Méthode	Rôle
Options addOption(String opt, boolean hasArg, String description)	Ajouter une option qui ne possède qu'un nom court
Options addOption(String opt, String longOpt, boolean hasArg, String description)	Ajouter une option qui possède un nom court et un nom long

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import org.apache.commons.cli.Options;
04.
05. public class App {
06.
07.   public static void main(final String[] args) {
08.     final Options options = new Options();
09.     options.addOption("n", true, "[nom] votre nom");
10.   }
11. }
```

La classe org.apache.commons.cli.OptionBuilder implémente le motif de conception monteur pour faciliter la configuration et l'obtention d'une instance de type Option. L'utilisation d'un OptionBuilder permet d'avoir un contrôle très précis sur la configuration de l'Option qui sera instanciée.

Méthode	Rôle
static Option create()	Obtenir l'instance à partir de la configuration courante
static Option create(char opt)	Obtenir l'instance à partir de la configuration courante et le nom court correspond au caractère fourni en paramètre
static Option create(String opt)	Obtenir l'instance à partir de la configuration courante et le nom court fourni en paramètre
static OptionBuilder hasArg()	Préciser que l'option possède un argument obligatoire
static OptionBuilder hasArg(boolean hasArg)	
static OptionBuilder hasArgs()	Préciser que l'option peut avoir plusieurs arguments dont le nombre n'est pas limité

static OptionBuilder hasArgs(int num)	Préciser que l'option peut avoir plusieurs arguments dont le nombre est fourni en paramètre
static OptionBuilder hasOptionalArg()	Préciser que l'option possède un argument facultatif
static OptionBuilder hasOptionalArgs()	Préciser que l'option peut avoir plusieurs arguments facultatifs
static OptionBuilder hasOptionalArgs(int numArgs)	Préciser que l'option peut avoir plusieurs arguments facultatif dont le nombre est fourni en paramètre
static OptionBuilder isRequired()	Préciser que l'option est obligatoire
static OptionBuilder isRequired(boolean newRequired)	
static OptionBuilder withArgName(String name)	Préciser le nom de l'argument de l'option
static OptionBuilder withDescription(String newDescription)	Préciser la description de l'option
static OptionBuilder withLongOpt(String newLongopt)	Préciser le nom long de l'option
static OptionBuilder withType(Object newType)	Préciser le type de la valeur de l'option
static OptionBuilder withValueSeparator()	Préciser que les valeurs de l'argument sont séparés par un caractère =
static OptionBuilder withValueSeparator(char sep)	Préciser le caractère de séparation des valeurs de l'argument

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import org.apache.commons.cli.OptionBuilder;
04. import org.apache.commons.cli.Options;
05.
06. public class App {
07.
08.   @SuppressWarnings("static-access")
09.   public static void main(final String[] args) {
10.     final Options options = new Options();
11.     options.addOption(OptionBuilder.withArgName("nom").hasArg()
12.       .withDescription("[nom] votre nom").create("n"));
13.   }
14. }
```

Il est possible de définir un groupe d'options qui contient un ensemble d'options parmi lesquelles une doit être utilisée.

Un groupe d'options est encapsulé dans un classe de type org.apache.commons.cli.OptionGroup.

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import org.apache.commons.cli.OptionBuilder;
04. import org.apache.commons.cli.OptionGroup;
05. import org.apache.commons.cli.Options;
06.
07. public class App {
08.
09.   @SuppressWarnings("static-access")
10.   public static void main(final String[] args) {
11.     final Options options = new Options();
12.     final OptionGroup groupe = new OptionGroup();
13.     groupe.setRequired(true);
14.     groupe.addOption(OptionBuilder.withDescription("Create").create("c"));
15.     groupe.addOption(OptionBuilder.withDescription("Read").create("r"));
16.     groupe.addOption(OptionBuilder.withDescription("Update").create("u"));
17.     groupe.addOption(OptionBuilder.withDescription("Delete").create("d"));
18.     options.addOptionGroup(groupe);
19.   }
20. }
```

Il est possible de définir des options utilisant le format des propriétés fournies à une JVM

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import org.apache.commons.cli.Option;
04. import org.apache.commons.cli.OptionBuilder;
05. import org.apache.commons.cli.Options;
06.
07. public class App {
08.
```

```
09.   @SuppressWarnings("static-access")
10.   public static void main(final String[] args) {
11.     final Options options = new Options();
12.     final Option property = OptionBuilder
13.         .withArgName("propriete=valeur")
14.         .hasArgs(2)
15.         .isRequired()
16.         .withValueSeparator()
17.         .withDescription("Assigner une valeur à la propriete")
18.         .create("D");
19.     options.addOption(property);
20.   }
21. }
```

Résultat :

```
1. java app -DmaPropriete=maValeur
```

La définition des options est la même quelque soit le parseur qui sera utilisé ultérieurement pour analyser les arguments passé en paramètre du programme.

99.2.2. L'analyse des options fournies en paramètres

Un parseur va utiliser la définition des options encapsulées dans une instance de type Options et les arguments passés au programme pour les analyser.

CLI propose plusieurs parseurs qui implémentent l'interface CommandLineParser en héritant de la classe Parser :

- BasicParser : analyseur basic qui impose un espace entre les options et leur argument.
- GnuParser : analyseur qui support le style d'options Gnu (support des options avec nom longs, l'argument peut être accolé à l'option en utilisant un caractère égal, ...)
- PosixParser : analyseur qui support le style d'options Posix (noms court des options, possibilité de concaténer plusieurs options, ...)

L'interface CommandLineParser définit deux méthodes :

Méthode	Rôle
CommandLine parse(Options options, String[] arguments)	Analyser les arguments par rapport aux options définies
CommandLine parse(Options options, String[] arguments, boolean stopAtNonOption)	Analyser les arguments par rapport aux options définies : le booléen permet de préciser si l'analyse se poursuit à la rencontre d'une option non définie

Les surcharges de la méthode parse() renvoient une instance de type CommandeLine qui encapsule les options et leurs valeurs extraites suite à l'analyse des arguments.

Exemple :

```
01. package com.jmdoudoux.test.cli;
02. import org.apache.commons.cli.CommandLineParser;
03. import org.apache.commons.cli.GnuParser;
04. import org.apache.commons.cli.Option;
05. import org.apache.commons.cli.OptionBuilder;
06. import org.apache.commons.cli.Options;
07. import org.apache.commons.cli.ParseException;
08.
09. public class App {
10.
11.   @SuppressWarnings("static-access")
12.   public static void main(final String[] args) {
13.     final Options options = new Options();
14.     final Option property = OptionBuilder
15.         .withArgName("propriete=valeur")
16.         .hasArgs(2)
17.         .isRequired()
18.         .withValueSeparator()
19.         .withDescription("Assigner une valeur à la propriete")
20.         .create("D");
21.     options.addOption(property);
22.     final CommandLineParser parser = new GnuParser();
23.     try {
24.       parser.parse(options, args, false);
25.     } catch (final ParseException exp) {
26.       System.err.println("Echec de l'analyse des options: " + exp.getMessage());
27.     }
28.   }
29. }
```

Résultat :

```
java App -DmaPropriete=maValeur -X
Echec de l'analyse des options: Unrecognized option: -X
```

Si une erreur est détectée lors de l'analyse par la méthode parse() alors elle lève une exception de type ParseException dont le message précise l'origine de l'anomalie.

L'exception ParseException possède plusieurs exceptions filles qui permettent d'obtenir des informations spécifiques :

- AlreadySelectedException : levée si plusieurs options d'un même groupe sont fournies en argument
- MissingArgumentException : levée si un argument obligatoire est manquant
- MissingOptionException : levée si une ou plusieurs options obligatoires sont manquantes
- UnrecognizedOptionException : levée si une option passée en argument n'est pas définies les options

99.2.3. L'obtention des options et de leurs valeurs

La classe CommandLine encapsule les options et les valeurs extraites suite à l'analyse des arguments passés à l'application.

Elle possède plusieurs méthodes qui permettent d'obtenir les informations qu'elle encapsule :

Méthode	Rôle
List getArgList()	
String[] getArgs()	
Object getOptionObject(char opt)	Retourner le type de l'option dont le nom court est passé en paramètre
Object getOptionObject(String opt)	Deprecated. Il est préférable d'utiliser la méthode getParsedOptionValue(String)
Properties getOptionProperties(String opt)	
Option[] getOptions()	Retourner un tableau des options
String getOptionValue(char opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
String getOptionValue(char opt, String defaultValue)	Retourner l'argument de l'option dont le nom court est passé en paramètre ou la valeur par défaut fournie en paramètre si l'argument n'est pas défini
String getOptionValue(String opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
String getOptionValue(String opt, String defaultValue)	Retourner l'argument de l'option dont le nom court est passé en paramètre ou la valeur par défaut fournie en paramètre si l'argument n'est pas défini
String[] getOptionValues(char opt)	Retourner un tableau des valeurs de l'option dont le nom court est passé en paramètre
String[] getOptionValues(String opt)	Retourner un tableau des valeurs de l'option dont le nom court est passé en paramètre
Object getParsedOptionValue(String opt)	Retourner l'argument de l'option dont le nom court est passé en paramètre
boolean hasOption(char opt)	Déterminer si l'option est définie ou non
boolean hasOption(String opt)	Déterminer si l'option est définie ou non
Iterator iterator()	Retourner un Iterator sur les options

Exemple :

```
01. package com.jmdoudoux.test.cli;
02.
03. import java.util.Arrays;
04. import org.apache.commons.cli.CommandLine;
05. import org.apache.commons.cli.CommandLineParser;
06. import org.apache.commons.cli.GnuParser;
07. import org.apache.commons.cli.Option;
08. import org.apache.commons.cli.OptionBuilder;
09. import org.apache.commons.cli.Options;
10. import org.apache.commons.cli.ParseException;
11.
12. public class App {
13.
14.   @SuppressWarnings("static-access")
15.   public static void main(final String[] args) {
16.     final Options options = new Options();
17.     final Option property = OptionBuilder
18.         .withArgName("propriete=valeur")
19.         .hasArgs(2)
20.         .isRequired()
21.         .withValueSeparator()
```

```
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
    final CommandLineParser parser = new GnuParser();
    try {
      final CommandLine line = parser.parse(options, args);
      if (line.hasOption("D")) {
        System.out.println(line.getOptionValue("D"));
        System.out.println(Arrays.deepToString(line.getOptionValues("D")));
      }
    } catch (final ParseException exp) {
      System.err.println("Echec de l'analyse des options: " + exp.getMessage());
    }
  }
}
```

Résultat :

```
java App -DmaPropriete=maValeur
maPropriete
[maPropriete, maValeur]
```

99.2.4. L'affichage d'une aide sur les options

La classe HelpFormatter permet de gérer un résumé des options qui pourra être affiché à l'utilisateur.

Elle possède plusieurs méthodes pour configurer les options de formatage et obtenir le résultat.

Plusieurs surcharges de la méthode printHelp() permettent d'afficher sur la sortie standard ou sur un Writer un résumé des options utilisables.

Plusieurs surcharges de la méthode printUsage() permettent d'afficher sur la sortie standard ou sur un Writer un résumé de l'utilisation des options.

Exemple :

```
package com.jmdoudoux.test.cli;

import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.OptionBuilder;
import org.apache.commons.cli.Options;

public class App {

  @SuppressWarnings("static-access")
  public static void main(final String[] args) {
    final Options options = new Options();
    final Option property = OptionBuilder
        .withArgName("propriete=valeur")
        .hasArgs(2)
        .isRequired()
        .withValueSeparator()
        .withDescription("Assigner une valeur à la propriete")
        .create("D");
    options.addOption(property);
    final HelpFormatter formatter = new HelpFormatter();
    formatter.printHelp("App", options, true);
  }
}
```

Résultat :

```
usage: App -D <propriete=valeur>
 -D <propriete=valeur>   Assigner une valeur à la propriete
```