

# Présentation de Java SE 6

par [F. Martini \(adiGuba\)](#)

Date de publication : 11/12/2006

Dernière mise à jour :

Alors que Java 5.0 s'annonçait comme une révolution, en apportant un grand nombre de modifications dans le langage, le nouvel opus de Java se présente plus serein et mature. Pas de révolution pour Java SE 6, mais de vraies évolutions afin de préparer le futur du langage et son ouverture aux autres langages.

Avant propos...

Java Standard Edition 6

I - Quoi de neuf docteur ?

JSR 223 : Scripting for the Java Platform

JSR 199 : Java Compiler API

JSR 269 : Pluggable Annotation-Processing API

JSR 250 : Common Annotations

JSR 202 : Class File Specification Update

JSR 221 : JDBC 4.0 API Specification

Les JSRs concernant XML et Web Services

II - Et les APIs existantes ?

AWT et l'intégration au système

Swing - Les principales nouveautés

L'API de Collection

Entrées/Sorties et Réseaux

Localisation et internationalisation

JVMTI (JVM Tool Interface)

Divers

Pour aller plus loin...

## Avant propos...

Je tiens à remercier l'équipe de rédacteurs Java de [developpez.com](#) pour leurs conseils avisés...

En particulier [wichtounet](#), [christopheJ](#), [loan](#) et [vbrabant](#).

## Java Standard Edition 6

La prochaine version de **Java** ne devrait plus tarder à être disponible en version finale. Il est donc intéressant de prendre le temps de découvrir ce que nous apportera ce nouvel opus. Outre un nouveau changement de nom et de numérotation, ce qui devient presque une tradition puisqu'on est passé de **J2SE 1.4.2** à **J2SE 5.0** et que l'on a désormais droit à **Java SE 6** (on se demande ce qu'ils nous préparent pour **Java 7** !), cette nouvelle version a essayé d'apporter des solutions dans six grands domaines :

- **Compatibilité ascendante.**

Bien que cela puisse paraître évident pour beaucoup de monde, l'accent est mis sur la compatibilité ascendante. Bien que la plateforme soit mature, elle continue d'évoluer et continuera d'évoluer sans cesse, mais cela ne remet nullement en cause la compatibilité des programmes. N'importe quel programme qui fonctionnait sur une version précédente de la plateforme Java devrait fonctionner de la même manière avec **Java SE 6**.

- **Simplification du développement.**

**Java 5.0** a apporté des améliorations significatives avec l'introduction de nouveaux types et syntaxes, comme les types paramétrés (generics), les annotations, les énumérations ou encore les boucles for-étendus. Il reste toutefois encore beaucoup de travail pour simplifier la vie du développeur, en particulier en ce qui concerne l'accès aux bases de données, le support des langages de script, des technologies fondamentales telles que la compilation ou le traitement des annotations, ou encore un meilleur support des EDI et autres outils de développement.

- **Diagnostic, surveillance et gestion.**

La plateforme Java est utilisée en production pour des applications critiques. **Java 5.0** avait apporté de nombreux progrès avec l'introduction de **JMX** (Java Management Extensions) et **JVMTI** (Java Virtual Machine Tools Interface) mais d'autres améliorations sont cependant nécessaires dans ce domaine.

- **Retour du "Desktop".**

Les applications riches sont revenues sur le devant de la scène et les développeurs commencent à atteindre les limites des clients légers basés sur les navigateurs. **Java** est une alternative possible, mais elle souffre toujours de problèmes d'intégrations au sein du système de l'utilisateur final et certains composants souffrent de quelques gros défauts qui devront être comblés.

- **XML et Web Services.**

**Java 5.0** devait originellement proposer un ensemble d'outils facilitant la création et l'utilisation des Web Services. Malheureusement cela a pris plus de temps que prévu et cela n'a pas pu être intégré dans la spécification, et pendant ce temps XML et les Web Services ont pris de plus en plus d'importance pour beaucoup de membres de la communauté.

- **Transparence dans l'évolution.**

La communauté a exprimé son désir de transparence dans l'évolution de la plateforme Java. Beaucoup voulaient pouvoir passer en revue et tester ces nouvelles spécifications alors même qu'elles étaient toujours en progrès, et devenir plus impliqué en contribuant à des améliorations ou des corrections de bugs. Le processus de développement de cette nouvelle version a donc été un des plus ouverts via son site communautaire : <https://jdk6.dev.java.net/>

## I - Quoi de neuf docteur ?

Afin de déterminer les principales nouveautés de cette nouvelle mouture, il faut se tourner vers les JSR associées à cette version. Les JSRs (Java Specification Requests) représentent le processus normalisé via lequel sont définies les différentes spécifications des plateformes Java (Standard, Entreprise et Mobile).

Pour chaque JSR, un groupe de travail est créé. Il est composé de personnes provenant des différents acteurs du monde Java (comme la **fondation Apache**, **IBM**, **BEA Systems**, **Borland** et bien d'autres), qui coopèrent afin de définir ces spécifications. Ces dernières sont soumises à un vote afin d'être acceptées et intégrées dans le langage.

Avant de commencer, je tiens à préciser que ce document ne se veut pas exhaustif du tout, mais simplement une présentation brèves des nouveautés que cela nous apportera. En effet, étant donné la taille (et la complexité) des différentes spécifications, l'étude approfondie de chacune de ces JSRs pourrait faire l'objet de plusieurs articles. Cet article se contente donc de les citer et de les décrire brièvement. Pour plus de détails je vous conseille vivement de consulter les spécifications disponibles sur la page de chaque JSR.

## JSR 223 : Scripting for the Java Platform

La **JSR 223** définit un framework permettant d'utiliser n'importe quel langage de script au sein d'une application Java.

Elle permet ainsi de connaître la liste des interpréteurs disponible sur la machine virtuelle, d'invoquer des scripts depuis une application Java (et vice-versa), de rendre les objets Java visibles et utilisables par ces scripts.

Par exemple, le code suivant permet grâce à l'interpréteur **JavaScript**, d'exécuter un script et de récupérer le résultat :

```
int intValue = 5;

ScriptEngineManager manager = new ScriptEngineManager();
// On récupère l'interpréteur de script JavaScript
ScriptEngine engine = manager.getEngineByName("JavaScript");
// On crée une association pour la variable Java 'intValue'
engine.getBindings(ScriptContext.ENGINE_SCOPE).put("intValue", intValue);
// Evaluation d'un script :
Object result = engine.eval(" ( 15 + intValue ) / 2 ");
// Affichage du résultat :
System.out.println("Résultat : " + result);
```

Ce qui nous donnera comme résultat :

```
Résultat : 10.0
```

Toutefois, il faut savoir que les spécifications n'imposent pas le support d'un langage de script en particulier. Ainsi chaque machine virtuelle peut proposer le support du/des langage(s) de son choix, et il faudra utiliser l'API pour vérifier la présence de l'interpréteur de script. Malgré cela, la JVM de Sun proposera d'office le support de **JavaScript** via l'interpréteur **rhino** de la **fondation Mozilla**, il y a donc de fortes chances que ce dernier se retrouve sur la plupart des JVM.

A noter également l'existence du projet "scripting" qui se chargera d'implémenter et de référencer les interpréteurs de script compatibles avec cette JSR. Elle comporte déjà plus de 20 langages de script dont **Python**, **Ruby**, **BeanShell** ou encore **Groovy** : <https://scripting.dev.java.net/>

Cette JSR, représentée par le package **javax.script**, permet donc théoriquement l'utilisation de n'importe quel langage de script au sein de vos applications Java, et offre ainsi un large panel de nouvelles perspectives.

Pour plus de détails sur cette JSR, vous pouvez télécharger les spécifications sur la page de la JSR : <http://jcp.org/en/jsr/detail?id=223>

## JSR 199 : Java Compiler API

Cette API définit un service permettant aux programmes Java d'invoquer un compilateur Java directement dans une application Java, et de récupérer toutes les informations retournées par le compilateur (erreurs, warnings, messages, etc.).

Cette JSR ne devrait pas vraiment concerner directement le développeur "de base", mais cible principalement les éditeurs d'outils Java :

- Les implémentations de moteur **JSP** pourront l'utiliser afin de compiler ces dernières sans invoquer de programme externe. Les premiers tests avec **GlassFish** (l'implémentation de référence de **Java EE 5**) ont donné des temps de compilations trois fois plus rapides (toute la compilation s'effectue en mémoire sans opération d'entrée/sortie).
- Les IDE pourront utiliser le compilateur directement, en utilisant une API standardisée (ce qui n'était pas le cas jusqu'à maintenant).
- D'autres outils ou API auront la possibilité de générer des fichiers **\*.class** via la génération et la compilation de code source à la volée.

Son API utilise le package **javax.tools** et vous trouverez ses spécifications sur la page de la JSR : <http://jcp.org/en/jsr/detail?id=199>

## JSR 269 : Pluggable Annotation-Processing API

Cette API permet de traiter les annotations lors de la compilation des codes sources, en interaction avec le compilateur. Il ne s'agit pas d'une nouveauté en soit, puisqu'un tel mécanisme était présent dans **Java 5.0** avec **APT** (Annotation Processing Tool). Toutefois, son API et les possibilités offertes ont été étendues et standardisées.

Cette API autorisera l'écriture de "bibliothèques intelligentes" qui communiqueront avec le compilateur afin de signaler des erreurs d'utilisation ou encore de générer dynamiquement des classes et/ou des fichiers de configuration.

Elle utilise les packages **javax.annotation.processing** (pour le traitement des annotations à l'exécution) et **javax.lang.model.\*** (qui représente le code lors de sa compilation).

Pour plus de détail : <http://jcp.org/en/jsr/detail?id=269>

## JSR 250 : Common Annotations

Cette JSR vient définir cinq nouvelles annotations standard qui pourront également être utilisées par la plateforme entreprise (**Java EE**).

- **@Generated** servira à marquer les codes (ou les portions de codes) qui ont été générés par un outil, afin de les différencier de ceux qui ont été écrits par un développeur.
- **@PostConstruct** et **@PreDestroy** sont liées à la persistance des données, et permettent respectivement de définir une méthode qui sera appelée soit après la construction de l'objet (et éventuellement après l'injection

de données), soit juste avant sa destruction.

- **@Resource** permet d'associer des ressources à des composants (classes, attributs ou méthodes), qui seront automatiquement injectées par les outils de déploiements. L'annotation **@Resources** quant à elle permettra simplement d'associer plusieurs ressources à un même composant.

Encore une fois, cette JSR n'est pas directement destinée aux développeurs mais aux outils divers, qui devront les utiliser pour simplifier la vie du développeur.

Le détail des spécifications est librement téléchargeable : <http://jcp.org/en/jsr/detail?id=250>

## JSR 202 : Class File Specification Update

Le format des fichiers \*.class a été mis à jour afin de supporter la "split verification", une architecture déjà présente dans la plateforme mobile de Java (Java ME), et qui apporte des avantages inhérents aux performances, aussi bien en terme d'espace utilisé (90%) que du temps de chargement de ces dernières (approximativement 2 fois plus rapide). Ce schéma est également plus simple et plus robuste, et facilitera les évolutions futures de la plateforme.

Bien entendu, tout cela est transparent pour le développeur, et la compatibilité ascendante reste d'actualité.

Toutes les informations sur ce nouveau format sont librement téléchargeables : <http://jcp.org/en/jsr/detail?id=202>

## JSR 221 : JDBC 4.0 API Specification

Les spécifications de l'**API JDBC 4.0** viennent apporter un vent de fraîcheur sur l'API d'accès aux bases de données de Java, en utilisant les facilités du langage apportées par Java 5.0 (Generics et Annotations en tête). Elle met l'accent sur les éléments suivant :

- Chargement automatique des drivers JDBC en utilisant le mécanisme de fournisseur de service.
- Utilisation d'annotations permettant de manipuler les tables facilement via un mapping object/relationnel très simple. Il est désormais possible d'exécuter des requêtes XML en écrivant un minimum de code Java.
- Support du type **SQL ROWID**, qui donne la possibilité de stocker l'index d'une ligne d'une table de la base de données.
- Amélioration du support des **BLOB** (Binary Large Object) et **CLOB** (Character Large Object).
- Support du type **SQL XML** défini dans le standard **SQL 2003**.
- Le gestionnaire de connexion a été amélioré et permet de détecter les connexions devenues invalides.
- La hiérarchie des exceptions a été revue, et la classe **SQLException** possède désormais deux classes filles de base dont hériteront toutes les exceptions liées à JDBC. Ainsi **SQLTransientException** représentera les exceptions "passagères" (comme les problèmes de connexion, les timeouts). A l'inverse, les exceptions qui hériteront de **SQLNonTransientException** indiqueront un problème plus grave lié à une erreur irrémédiable qui devra être corrigée (argument invalide, erreur de syntaxe, etc.). De plus, toutes les **SQLException** sont désormais itérables et peuvent donc être utilisées dans une boucle for afin de dépiler la chaîne d'exception :

```
catch (SQLException e) {
    for (Throwable cause : e) {
        cause.printStackTrace();
    }
}
```

Les spécifications de **JDBC 4.0** sont bien sûr également disponibles : <http://jcp.org/en/jsr/detail?id=221>

## Les JSRs concernant XML et Web Services

Comme prévu, **Java SE 6** sera fortement axé vers XML et les Web Services, avec ni plus ni moins que 5 JSRs sur le sujet, brièvement présentées ici :

- **JSR 105 : XML Digital-Signature API**

Cette API, représentée par le package **javax.xml.crypto**, est une implémentation de la norme **XML Digital-Signature** du **W3C** (<http://www.w3.org/TR/2002/REC-xmlsig-core-20020212/>) et est une clef importante de la sécurité des Web Services. De plus elle est requise par d'autres JSR de cette section.

Plus d'info : <http://jcp.org/en/jsr/detail?id=105>

- **JSR 173 : Streaming API for XML (StAX)**

**StAX** est une API permettant d'exploiter les documents XML tout comme **DOM** ou **SAX** dont elle reprend les principaux avantages. **StAX** lit le document de manière linéaire afin de ne pas encombrer la mémoire (comme **SAX**), par contre la lecture du document n'est pas événementielle mais c'est l'application qui détermine les éléments à lire (à l'instar de **DOM**). Bref **StAX** prend le meilleur des deux...

Plus d'info : <http://jcp.org/en/jsr/detail?id=173>

- **JSR 181 : Web-Services Metadata for the Java Platform**

Cette JSR définit un ensemble d'annotations permettant de décrire facilement des Web-Services au niveau du code source. Ces annotations détermineront comment les serveurs d'applications déploieront ces Web-Services.

Plus d'info : <http://jcp.org/en/jsr/detail?id=181>

- **JSR 222 : Java Architecture for XML Binding (JAXB) 2.0**

Il s'agit d'une révision majeure des spécifications de **JAXB 1.1**, l'API permettant de générer des classes Java à partir de schéma XML et inversement. Cette version met l'accent (entre autres) sur le support de la norme des **XML Schema**.

Plus d'info : <http://jcp.org/en/jsr/detail?id=221>

- **JSR 224 : Java API for XML-Based Web Services (JAX-WS) 2.0**

Cette spécification étend **JAX-RPC 1.0** afin de simplifier le développement, de mieux s'intégrer avec **JAXB 2.0** et d'apporter le support des derniers standards (**SOAP 1.2**, **WSDL 2.0** et **WS-I Basic Profile 1.1**).

Plus d'info : <http://jcp.org/en/jsr/detail?id=224>

## II - Et les APIs existantes ?

En plus de ces nouvelles spécifications, les APIs existantes ont également eut leurs lots de nouveautés et de modifications.

### AWT et l'intégration au système

La mauvaise intégration des applications Java dans le système d'exploitation est un des points noirs des applications Java. Cette nouvelle version comble un peu ce déficit, notamment en ce qui concerne l'intégration au système :

La classe **java.awt.SystemTray** permet d'ajouter une icône dans la zone de notification du système (la plupart du temps il s'agit des icônes à coté de l'horloge) afin d'interagir avec l'utilisateur (clic, menu déroulant, affichage d'infobulle évoluée).

La classe **java.awt.Desktop** permet d'intégrer l'application au bureau du système d'exploitation, et permet trois principales actions :

- Ouvrir le navigateur par défaut sur une page précise (spécifiée par une **URI**).
- Ouvrir la fenêtre de saisie d'email du client de courrier électronique par défaut (éventuellement en y renseignant les différents champs via une **URI** de type **mailto:**).
- Utiliser l'association de fichier du système d'exploitation afin de lancer le programme associé pour effectuer des actions de base (ouvrir, éditer et imprimer).

Les boites de dialogues bénéficient d'un nouveau système de modalité plus élaboré. Ainsi on distingue désormais quatre types de modalité représentés par l'énumération **ModalityType** :

- **MODELESS** : La boite de dialogue n'est pas modale, elle ne bloquera donc aucune autre fenêtre.
- **APPLICATION\_MODAL** : La boite de dialogue bloquera toutes les autres fenêtres et boites de dialogues de l'application (sauf ses éventuelles fenêtres filles). Il s'agit du comportement actuel des boites de dialogues modales.
- **DOCUMENT\_MODAL** : La boite de dialogue bloquera toutes les fenêtres de la même hiérarchie, c'est à dire toutes ses fenêtres parentes, mais ne bloquera pas les autres fenêtres de l'application.
- **TOOLKIT\_MODAL** : La boite de dialogue bloquera toutes les fenêtres du même toolkit (sauf ses éventuelles fenêtres filles). Ainsi par exemple toutes les applets d'un navigateur sont exécutées dans le même toolkit.

Il est également possible de définir, indépendamment pour chaque fenêtre, des exceptions à ces règles (représentées par l'énumération **ModalExclusionType**) :

- **NO\_EXCLUDE** : La fenêtre ne bénéficiera d'aucune exclusion et sera bloquée selon les règles en vigueur.
- **APPLICATION\_EXCLUDE** : La fenêtre ne sera pas bloquée par les boites de dialogues de type **APPLICATION\_MODAL**.
- **TOOLKIT\_EXCLUDE** : La fenêtre ne sera pas bloquée par les boites de dialogues de type **TOOLKIT\_MODAL**.

A première vue ces fonctions n'ont rien de révolutionnaire et pourraient même surprendre les développeurs non-java puisqu'il s'agit d'éléments qui peuvent sembler basiques. Il s'agit pourtant de notions qui ne sont pas forcément présentes sur tout les systèmes d'exploitations, et qui n'étaient donc pas intégrées dans l'API standard afin de ne pas casser la sacro-sainte loi "*Write once, run anywhere*" ("*Ecrire une fois, exécuter partout*").

Afin de ne pas trop mettre à mal cette règle, et étant donné que ces fonctionnalités peuvent carrément être

absentes du système d'exploitation, il est à la charge du développeur de vérifier dynamiquement le support de ces éléments via des méthodes adéquates, ainsi :

- Avant d'ajouter une icône dans la zone de notification, il faut vérifier que cela est bien supporté par le système avec la méthode **SystemTray.isSupported()**.
- Avant d'utiliser la classe **Desktop**, il faut vérifier si elle est supportée avec la méthode **Desktop.isDesktopSupported()**, puis pour chacune des actions possibles avec la méthode **Desktop.isSupported(Action)**.
- Avant d'utiliser un type de modalité ou d'exclusion, il faut vérifier que celui-ci soit supporté par le système avec les méthodes **Toolkit.isModalityTypeSupported(ModalityType)** et **Toolkit.isModalExclusionTypeSupported(ModalExclusionType)**.

Si la fonctionnalité est utilisée alors qu'elle n'est pas supportée, le comportement varie selon les cas. Ainsi pour les classes **Desktop** et **SystemTray**, une exception sera remontée, alors que lors de l'utilisation des types de modalité ou d'exclusions une valeur par défaut sera utilisée à la place (respectivement **MODELESS** et **NO\_EXCLUDE**). Ainsi, il est à la charge du programmeur de proposer une méthode alternative si besoin.

Il est intéressant de noter que le même principe a été appliqué pour la méthode **setAlwaysOnTop()** introduite dans **Java 5.0** avec l'apparition des méthodes **Window.isAlwaysOnTopSupported()** et **Toolkit.isAlwaysOnTopSupported()**.

Note : Toutes ces fonctionnalités concernent le package **java.awt**, mais elles s'appliquent également aux applications **Swing** (n'oublions pas que **Swing** "hérite" d'**AWT**).

## Swing - Les principales nouveautés

**Swing** a également subi pas mal d'améliorations, dont voici les principales :

- Tout d'abord les LookAndFeels natif **Windows** et **GTK** ont subi de nettes améliorations. Ils ne se contentent plus de recopier l'apparence du système, mais utilisent leurs API systèmes respectifs afin de récupérer les divers éléments de l'interface. Ils s'intègrent donc parfaitement dans l'environnement de l'utilisateur quel que soit son thème. Bien sûr cela fonctionne parfaitement sous **Windows Vista**.
- La classe **java.awt.SplashScreen** permet de gérer un "splash screen" natif au démarrage de l'application en spécifiant une image (via le manifest du jar ou une option de la ligne de commande). Ce dernier sera automatiquement détruit lorsque la première fenêtre de l'application sera ouverte.
- L'API destinée aux **LayoutManager** et aux outils d'interface graphique permet désormais de connaître la ligne de base et les espacements préférés des composants, afin de mieux les disposer. De plus la famille des **LayoutManager** standard est agrandie par l'arrivée de **javax.swing.GroupLayout**, le manager utilisé par **Matisse**, le GUI-Builder de **NetBeans**, qui permet de grandement simplifier la création d'interfaces graphiques **Swing**.
- Tous les composants textes de **Swing** (héritant de **JTextComponent**) bénéficient désormais d'un support évolué de l'impression qui leur manquait cruellement.
- Il est dorénavant possible d'utiliser n'importe quel composant **Swing** en guise d'onglets sur les **JTabbedPane** grâce à la méthode **setTabComponentAt()**.
- Les données des **JTables** pourront être triées ou filtrées simplement via **javax.swing.RowSorter** et **javax.swing.RowFilter**.
- Le **drag and drop** (glisser-déposer) a été grandement amélioré et simplifié afin de permettre d'utiliser plusieurs modes de fonctionnement via l'énumération **javax.swing.DropMode**.
- La classe **SwingWorker** permettant une gestion facile des traitements sans bloquer l'interface graphique est enfin intégrée en standard (cette classe faisait partie des tous premiers tutoriels officiels sur **Swing**).
- A ce propos, l'affichage via le double-buffer de **Swing** comblera désormais en partie le bug du rectangle gris ("gray rect") lorsque vous bloquez le thread d'affichage (**EDT**) : la fenêtre sera quand même complètement dessinée et n'affichera plus l'affreux rectangle gris (mais bien sûr votre interface sera quand même figée).

- La qualité de l'affichage des textes a été améliorée, notamment en utilisant la configuration système en ce qui concerne l'anti-aliasing.
- Le pipeline de rendu **OpenGL** (intégré avec **Java 5.0**) a été revu afin d'utiliser un mode **STR** (Single-Threaded Rendering).

## L'API de Collection

Du côté de l'API de Collection, on note l'arrivée de cinq nouvelles interfaces :

- **Deque** : une queue à double sens, qui étend l'interface **Queue** de **Java 5.0** afin de supporter les insertions/suppressions à ses deux extrémités.
- **BlockingDeque** : une **Deque** avec des opérations bloquantes lors de l'ajout et suppression d'élément.
- **NavigableSet**, qui étend **SortedSet** afin de permettre une navigation en ordre croissant ou décroissant et des méthodes de recherches évoluées.
- **NavigableMap**, qui étend **SortedMap** afin de permettre une navigation en ordre croissant ou décroissant et des méthodes de recherches évoluées.
- Enfin, **ConcurrentNavigableMap** permet les mêmes possibilités tout en étant naturellement sécurisée en environnement multithread.

Cela s'accompagne également de nouvelles implémentations concrètes :

- **ArrayDeque** : Une implémentation de **Deque** basé sur l'utilisation de tableau redimensionnable.
- **ConcurrentSkipListSet** et **ConcurrentSkipListMap**, implémentent respectivement **NavigableSet** et **ConcurrentNavigableMap**.
- **LinkedBlockingDeque** : Une implémentation de **BlockingDeque** basé sur une liste chaînée.

Sans oublier l'existence de deux implémentations de **Map.Entry** qui devrait faciliter l'écriture d'implémentation de **Map** personnalisée :

- **AbstractMap.SimpleEntry** est une implémentation de base.
- **AbstractMap.SimpleImmutableEntry** est une implémentation immuable (dont on ne peut pas modifier le contenu après la création).

A noter également la présence des méthodes **Arrays.copyOf()** et **Arrays.copyOfRange()** permettant de redimensionner/tronquer/copier des tableaux de tous types plus simplement qu'avec la méthode **System.arraycopy()** quelque peu complexe...

## Entrées/Sorties et Réseaux

- La classe **java.io.Console**, dont l'unique instance est accessible via la méthode **System.console()**, nous permet une gestion un peu plus fine de la console, comme la lecture de mot de passe sécurisé et l'accès aux objets **Reader/Writer** associés. On regrettera toutefois l'impossibilité de manipuler le curseur et/ou d'effacer certaine portion de texte.
- La classe **File** permet désormais d'obtenir plus d'informations concernant l'utilisation du système de stockage, avec les méthodes suivantes :

**getTotalSpace()** indique la taille totale de la partition (en bytes).

**getFreeSpace()** indique la taille disponible sur la partition (en bytes).

**getUsableSpace()** indique la taille utilisable sur la partition (en bytes). Ceci inclut la vérification des droits d'accès et d'éventuelles restrictions systèmes.

- De plus, il est désormais possible de modifier les attributs des fichiers (pour l'utilisateur courant ou pour tout le monde) via les méthodes **setWritable()**, **setReadable()** et **setExecutable()** de la classe **File**. En toute logique, la réussite de ces méthodes dépend du système d'exploitation.
- **java.net.IDN** apporte le support des noms de domaine internationalisé (**IDN**) au format Unicode et de leurs équivalents ASCII.
- L'interface **java.net.NetworkAddress** a été étendue afin de fournir plus d'informations sur le réseau (adresse de broadcast, masque réseau, adresse MAC, taille du MTU, état activé ou désactivé, etc.), notamment via l'interface **java.net.InterfaceAddress**.
- **Java 5.0** avait introduit la classe abstraite **java.net.CookieHandler** permettant de gérer les Cookies lors des connexions HTTP, mais ne proposait aucune implémentation par défaut. C'est désormais chose faite avec **java.net.CookieManager** qui propose une implémentation qui devrait suffire à la plupart des applications.

## Localisation et internationalisation

- *Pluggable locale* : La gestion de locale du JRE permet désormais l'installation de Locale non-supportée via un système de plugin.
- Les **ResourceBundles** ont bénéficié de quelques améliorations en proposant un dispositif permettant de contrôler la gestion et le comportement du cache, ou encore le support de n'importe quel type de formats en entrée (comme XML par exemple).
- La classe **java.text.Normalizer** permet la transformation de chaînes de caractères Unicode dans une forme composée ou décomposée, selon les standards de l'Unicode. En utilisant la même forme d'encodage, on peut manipuler les chaînes avec des caractères spéciaux ou des alphabets exotiques plus simplement. A titre d'exemple, le caractère **À** peut être représenté par la forme composée **\u00C1** (*A majuscule avec accent grave*) ou par la forme décomposée **\u0041\u0301** (*A majuscule combiné avec un accent grave*). En utilisant la même forme pour toutes les chaînes on facilite la comparaison de ce type de formation.
- Ajout du support du calendrier impérial chinois et de nombreuses nouvelles locales (zh\_SG, en\_MT, en\_PH, en\_SG, el\_CY, id\_ID, ga\_IE, ms\_MY, mt\_MT, pt\_BR, pt\_PT, et es\_US).

## JVMTI (JVM Tool Interface)

- Les outils d'analyses auront dorénavant la possibilité d'accéder au contenu de la mémoire heap via **JVMTI**.
- **Attach-on-demand** : **Java 5.0** avait apporté la possibilité de surveiller le comportement d'une JVM lorsqu'elle était lancée avec certains paramètres. Il sera désormais possible de surveiller n'importe quelle application Java quel que soit la manière dont elle a été démarrée (et donc de démarrer une surveillance sur une application sans la redémarrer).

## Divers

- Calcul à virgule flottante : ajout de méthodes recommandées par l'**IEEE 754**. Ainsi les classes **Math** et **StrictMath** se sont vu rajoutées les méthodes **scalb()**, **getExponent()**, **nextAfter()**, **nextUp()**, et **copySign()** pour les types **float** et **double**.
- Le package **java.util.concurrent** introduit dans **Java 5.0** a reçu quelques correctifs mineurs, comme l'ajout de nouvelles unités de temps dans la classe **TimeUnit** : **MINUTES**, **HOURS** et **DAYS**.
- La nouvelle classe **java.util.ServiceLoader** permet de gérer un système de service tel qu'il est déjà utilisé par plusieurs classes de l'API standard, en recherchant toutes les implémentations concrètes d'une interface (ou d'une classe abstraite) déclarée dans un des fichiers des répertoires **META-INF/services** de chacun des éléments du **CLASSPATH**. Ceci est utilisé par exemple pour rechercher automatiquement les drivers **JDBC**.
- On retiendra également le support des images au format GIF en écriture (jusqu'à présent seul le support en lecture était disponible en standard) qui est devenu possible avec l'expiration des brevets le concernant (le format est dorénavant dans le domaine public).

- Enfin la définition du **CLASSPATH** supporte désormais le wildcard **\*** qui représente toutes les archives **jar/zip** d'un répertoire. Ainsi si le **CLASSPATH** (via la ligne de commande, la variable d'environnement ou l'attribut du manifest du Jar) comporte par exemple **lib/\*** toutes les archives du répertoire **lib** seront ajoutées au Classpath de l'application, et il sera donc inutile de les nommer une à une.

## Pour aller plus loin...

Si le sujet vous intéresse et que vous voulez approfondir les choses, je ne peux que vous conseiller d'aller jeter un coup d'oeil aux différentes JSRs ou sur le site officiel :

- **Java™ SE 6 Release Notes - Features and Enhancements :**  
<http://java.sun.com/javase/6/webnotes/features.html>
- **JDK 6 Documentation :** <http://java.sun.com/javase/6/docs/>
- **Java Platform API Specification :** <http://java.sun.com/javase/6/docs/api/>

Et pourquoi pas de venir en parler avec nous sur le forum **Java** de **Developpez.com** :  
<http://www.developpez.net/forums/showthread.php?t=250242>