

Exécuter une application externe en Java



par Yann D'ISANTO ([Autres articles](#)) Frédéric Martini ([Autres articles](#))

Date de publication : 20/12/2006

Dernière mise à jour : 21/09/2007

Voici un petit guide des bonnes pratiques pour exécuter une application externe en Java.

- I - Introduction
- II - Lancer une application externe
 - II-A - La classe Runtime
 - II-B - La classe Process
- III - Communiquer avec l'application
 - III-A - Récupération des flux
 - III-B - Consommation des flux
- IV - Mise en pratique
- V - Runtime.exec() n'est pas un shell
 - V-A - Problématique
 - V-B - Solution
- VI - Remarques
 - VI-A - La classe ProcessBuilder
 - VI-B - JDIC et la classe Desktop
 - VI-C - Java 6 et la classe Desktop
 - VI-D - L'API Shell
- VII - Remerciements

I - Introduction

Il arrive fréquemment que l'on doive lancer une application externe depuis un programme Java. Java nous le permet, cependant beaucoup de personnes rencontrent des difficultés souvent dues à une méconnaissance de certains principes pourtant fondamentaux.

II - Lancer une application externe

II-A - La classe Runtime

L'exécution d'une application externe se fait grâce aux méthodes `exec()` de la classe **Runtime**. Chaque application Java possède une instance unique de la classe *Runtime* qui lui permet de s'interfacer avec son environnement.

L'instance se récupère avec la méthode statique `getRuntime()`.

```
Runtime runtime = Runtime.getRuntime();
```

Pour lancer votre application externe il vous suffit maintenant d'appeler l'une des six méthodes `exec()` de la classe *Runtime* et dont voici les déclarations :

```
public Process exec(String command);
```

Permet d'exécuter une ligne de commande dans un processus séparé.

```
public Process exec(String[] cmdarray);
```

Permet d'exécuter une commande avec ses arguments dans un processus séparé.

```
public Process exec(String[] cmdarray, String[] envp);
```

Permet d'exécuter une commande avec ses arguments dans un processus séparé en spécifiant des variables d'environnement.

```
public Process exec(String[] cmdarray, String[] envp, File dir);
```

Permet d'exécuter une commande avec ses arguments dans un processus séparé en spécifiant des variables d'environnement et le répertoire de travail.

```
public Process exec(String command, String[] envp);
```

Permet d'exécuter une ligne de commande dans un processus séparé en spécifiant des variables d'environnement.

```
public Process exec(String command, String[] envp, File dir);
```

Permet d'exécuter une ligne de commande dans un processus séparé en spécifiant des variables d'environnement et le répertoire de travail.

Remarque : Les variables d'environnement spécifiées doivent l'être selon le format *nom=valeur*.

Un point important est que si vous voulez lancer une application externe en lui passant des paramètres, il faut toujours passer par une des méthodes `exec()` attendant un tableau de *String*.

Même s'il est possible d'utiliser une des méthodes `exec()` attendant un simple *String* pour lancer une application avec des paramètres :

```
Runtime runtime = Runtime.getRuntime();  
runtime.exec("monappli param1 param2");
```

cela risque de poser des problèmes si l'un de vos paramètres contient un espace.

En effet, java utilise le caractère espace pour extraire les différents paramètres de la ligne de commande. Donc si vous avez un paramètre du genre "un paramètre avec des espaces", Java le comprendra comme cinq paramètres différents ("un", "paramètre", "avec", "des", "espaces").

C'est pourquoi il faut toujours utiliser un tableau de *String* pour passer des paramètres à une application externe.

```
Runtime runtime = Runtime.getRuntime();  
runtime.exec(new String[] { "monappli", "un paramètre avec des espaces", "param2" } );
```

Cette remarque est aussi valable si la commande elle-même contient des espaces.

```
Runtime runtime = Runtime.getRuntime();  
runtime.exec(new String[] { "C:\\Program Files\\MonAppli\\monappli.exe" } );
```

II-B - La classe Process

Comme vous l'avez sans doute remarqué, les différentes méthodes `exec()` renvoie un objet de type **Process**. Cette classe représente le processus de l'application externe et va nous permettre d'interagir avec lui. La classe *Process*, qui est abstraite, définit les 6 méthodes suivantes :

- la méthode **destroy()** qui permet de tuer le processus de l'application externe,
- la méthode **exitValue()** qui permet de récupérer la valeur de retour du processus de l'application externe,
- la méthode **getErrorStream()** qui permet de récupérer le flux d'erreur du processus de l'application externe,
- la méthode **getInputStream()** qui permet de récupérer le flux de sortie du processus de l'application externe,

- la méthode **getOutputStream()** qui permet de récupérer le flux d'entrée du processus de l'application externe,
- la méthode **waitFor()** qui met le thread courant en attente que le processus de l'application externe se termine.

III - Communiquer avec l'application

Remarque : pour cette partie il est nécessaire de connaître le fonctionnement des flux d'entrée/sortie en Java (cf ce [tutoriel sur le package java.io](#)).

III-A - Récupération des flux

Si besoin est, nous avons la possibilité de communiquer avec notre application externe au travers des trois flux récupérables par les méthodes `getErrorStream()`, `getInputStream()` et `getOutputStream()` de la classe `Process` :

- la méthode **`getErrorStream()`** permet de récupérer un `InputStream` représentant le flux d'erreur de l'application externe.
- la méthode **`getInputStream()`** permet de récupérer un `InputStream` représentant le flux de sortie de l'application externe.
- la méthode **`getOutputStream()`** permet de récupérer un `OutputStream` représentant le flux d'entrée de l'application externe.

Remarque : au premier abord il peut paraître étrange de récupérer un `InputStream` pour le flux de sortie standard. Cependant il faut bien se placer au niveau de l'application Java.

En effet, il s'agit de la sortie standard de l'application externe, l'application Java va lire ce flux qui est donc de son point de vue (en fait le notre) un flux d'entrée (idem pour le flux d'erreur). De même pour l'entrée standard de l'application externe, du point de vue de l'application Java il s'agit d'un flux de sortie puisqu'elle y écrit (d'où le `OutputStream`).

III-B - Consommation des flux

L'un des problèmes majoritairement rencontré est le fait que l'application externe semble se bloquer. Cela est souvent dû à une mauvaise gestion des flux.

En effet, les redirections d'E/S utilisent des buffers de taille limité (et dépendant du système hôte). Si les flux d'E/S ne sont pas traités par le programme appelant, le processus peut se retrouver bloqué. Pire encore : on peut facilement se retrouver dans un cas d'inter-blocage (le processus attend que le programme Java vide le buffer du flux afin de pouvoir continuer son exécution, alors que le programme Java attend que le processus fils se termine pour continuer son exécution, et donc les deux applications s'attendent mutuellement).

Enfin, et toujours pour éviter des inter-blocages, les différents flux doivent être traités **depuis des threads différents**, ce qui vient encore compliquer le tout.

```
Runtime runtime = Runtime.getRuntime();
final Process process = runtime.exec("monappli");

// Consommation de la sortie standard de l'application externe dans un Thread separé
new Thread() {
    public void run() {
        try {
```

```
BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line = "";
try {
    while((line = reader.readLine()) != null) {
        // Traitement du flux de sortie de l'application si besoin est
    }
} finally {
    reader.close();
}
} catch(IOException ioe) {
    ioe.printStackTrace();
}
}.start();

// Consommation de la sortie d'erreur de l'application externe dans un Thread separe
new Thread() {
    public void run() {
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(process.getErrorStream()));
            String line = "";
            try {
                while((line = reader.readLine()) != null) {
                    // Traitement du flux d'erreur de l'application si besoin est
                }
            } finally {
                reader.close();
            }
        } catch(IOException ioe) {
            ioe.printStackTrace();
        }
    }
}.start();
```

Chaque flux **doit** être traité, c'est à dire que le flux d'entrée doit recevoir des données puis être fermé, et les flux de sortie doivent être lus assez rapidement pour éviter de bloquer le buffer.

Le code n'est pas bien compliqué, mais il faut avouer qu'il est assez "pénible" à écrire, surtout à cause du fait que ces lectures/écritures doivent être effectuées depuis des threads différents. On peut toutefois simplifier cela en fermant directement les flux qui ne sont pas utilisés...

IV - Mise en pratique

La classe **ProcessLauncher (LGPL)** utilise les principes vus précédemment afin de lancer une application externe dans de bonnes conditions.

V - Runtime.exec() n'est pas un shell

V-A - Problématique

Il est important de bien comprendre que les diverses méthodes `exec()` de la classe `Runtime` permettent de lancer une **application** et non d'interpréter une ligne de commande ! C'est à dire que le programme appelé doit correspondre à un fichier exécutable, et que chacun des paramètres lui seront passés tel quel sans modification.

En effet, l'interprétation des lignes de commandes fait partie des fonctions du shell, et s'il est possible de l'appeler facilement grâce à la fonction `system()` du **C**, il faut malheureusement constater qu'il n'existe aucune équivalence en standard en Java ! Pourtant le shell apporte de nombreux avantages puisqu'il permet d'exécuter des commandes bien plus élaborées :

- La gestion des redirections avec `<`, `>`, `>>`, `2>`, `2>>` et `2>&1`.
- La gestion des pipes de processus avec `|` et des opérateurs booléens `&&` et `||`.
- La gestion des résolutions des variables d'environnements selon la norme du système (`%NAME%` sous Windows et `$NAME` sous les systèmes Unix).
- La gestion des commandes builtins du shell (echo, cd, etc.).
- Et plus globalement de toutes les spécificités du shell du système d'exploitation hôte, comme l'interprétation des meta-caractères des shell Unix (`*`, `?`, `\`, etc.)

Toutes ces fonctionnalités **ne sont pas utilisables** avec les méthodes `exec()`.

La raison est toute simple : l'appel de commande du shell gênerait à la portabilité de l'application (il faudrait gérer des commandes différentes selon le système cible).

Mais si l'intention est louable, elle est également très problématique car l'appel du moindre programme natif en Java peut devenir un vrai calvaire (ou presque).

V-B - Solution

On ne peut pas évaluer des lignes de commandes directement, mais il est possible d'appeler le programme représentant le shell système pour qu'il les évalue. Le problème étant que ce même shell dépend du système, c'est à dire par défaut `command.com` sous les **Windows 9x**, `cmd.exe` sous les **Windows NT**, `/bin/sh` sous les systèmes **Unix** et assimilés. Mais il peut également y avoir des shells "personnalisés", généralement via les variables d'environnements `%ComSpec%` sous Windows, ou `$SHELL` sous Unix...

Ces shells acceptent tous un paramètre `/C` (sous Windows) ou `-c` (sous Unix) qui permet d'interpréter une ligne de commande complexe.

Ce n'est pas bien compliqué à mettre en oeuvre, si ce n'est qu'il faut déterminer le shell système pour tenter d'avoir un tant soit peu de portabilité (même si l'appel de programme externe ou de ligne de commande nuit déjà à la portabilité).

Voici un exemple sous Windows renvoyant le résultat d'un **dir** dans un fichier

```
Runtime runtime = Runtime.getRuntime();
String[] args = { "cmd.exe", "/C", "dir C:\\ >fichier.txt" };
final Process process = runtime.exec(args);

// Et tout le traitement des flux d'E/S vu plus haut
```

Et voici l'équivalent pour les systèmes Linux/Unix (commande **ls**) :

```
Runtime runtime = Runtime.getRuntime();
String[] args = { "/bin/sh", "-c", "ls / >fichier.txt" };
final Process process = runtime.exec(args);

// Et tout le traitement des flux d'E/S vu plus haut
```

Mais il faut avouer que c'est quand même un peu casse pied de devoir faire tout cela rien que pour lancer une application ou une ligne de commande.

Pour vous éviter ce travail, vous pouvez utiliser l'API **Shell**, développée par Frédéric Martini, abordée plus loin dans ce tutoriel.

VI - Remarques

VI-A - La classe ProcessBuilder

Depuis Java 5, nous avons à notre disposition la classe **ProcessBuilder** qui permet entre autres de fusionner les flux de sortie et d'erreur du *Process*.

VI-B - JDIC et la classe Desktop

L'API **JDIC** possède une classe très intéressante pour notre sujet. Il s'agit de la classe **Desktop** qui permet notamment d'ouvrir un fichier avec l'application qui lui est associée par le système. A noter cependant que vous n'aurez aucun contrôle sur le processus lancé et que vous ne pourrez pas communiquer avec lui.

Pour plus d'information sur l'API JDIC, reportez vous au tutoriel **JDesktop Integrated Components**.

VI-C - Java 6 et la classe Desktop

Depuis Java 6, la classe **Desktop** a été intégrée dans l'API standard et est quasiment identique à sa soeur de l'API JDIC.

VI-D - L'API Shell

L'API Shell permet de simplifier l'exécution de programme et de ligne de commande depuis Java, vous pouvez donc profiter simplement des fonctionnalités du shell système.

Vous trouverez de plus amples informations sur l'API Shell dans [ce billet de Frédéric Martini](#).

VII - Remerciements

Je voudrais remercier adiGuba pour sa contribution ainsi que Wichtounet, Maxoo, Valered et Afrikha pour leur aide et leurs corrections.

