



Les promesses de Java 7

La prochaine version de Java – nom de code "dolphin" – sera la septième selon la numérotation "produit" de SUN. Peu de choses ont été fixées pour cette nouvelle version. Il n'existe même pas de JSR la concernant ! Quand pourrons-nous en disposer ? Mystère ... Et pourtant, l'information serait très appréciée, car il semblerait bien qu'il s'agisse d'une "grosse" version, à l'instar de ce que fut, en 2005, le JDK 5.0 avec les annotations et les generics.

Mais voilà, tout doit être mis au conditionnel, et un usage intensif en est fait dans cet article. S'il est possible de glaner des informations sur les blogs des experts impliqués dans les diverses API qui pourraient être embarquées dans le JDK 7.0, il est impossible aujourd'hui de trouver LE document de référence qui indique ce que contiendra réellement cette version.

Comme le JDK 5.0, de notables évolutions de la syntaxe de Java seraient prévues. On trouve d'abord quelques ajustements mineurs facilitant çà et là l'écriture de code. Mais la principale nouveauté attendue est l'introduction des "closures" (appelées parfois "expressions lambda") dans le langage Java. Notons que les "closures" existent sur .NET à partir de la version 3.0.

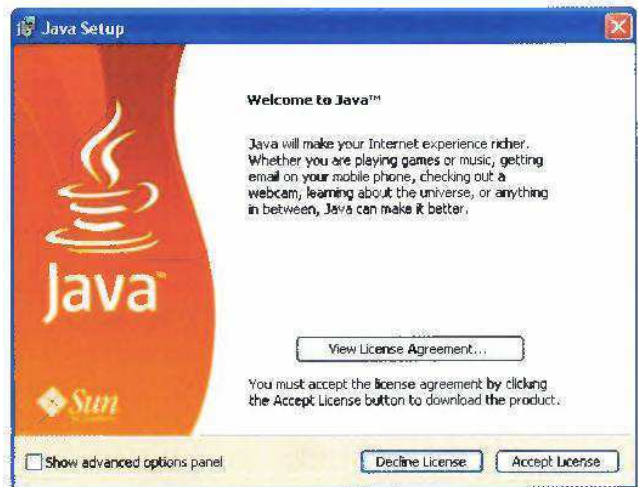
Une autre nouveauté qui risque de faire date est la définition de modules Java (Java Module), sorte de super JAR (Java Archive), permettant de faciliter et fiabiliser la gestion des bibliothèques de classes.

Swing n'est pas oublié : SUN continue à croire au destin de Java sur le poste client, comme le montrait déjà l'initiative JavaFX. Le JDK 7 devrait embarquer les JSR 295 (Bean Bindings), JSR 296 (Swing Framework) et JSR 303 (Bean Validation). Un peu plus d'API et donc de complexité pour un mécanisme graphique déjà très riche (trop riche ?). Cela facilitera probablement l'écriture d'applications desktop, mais nécessitera un apprentissage supplémentaire. Beaucoup d'équipes de développeurs Swing risquent de se tourner vers des solutions globalement plus simples comme JavaFX. On peut certes objecter qu'un framework standard est de toute façon préférable à un framework "maison" bâti au dessus de Swing. Il existe bien d'autres projets qui pourraient (le conditionnel encore !) faire partie de Java 7. Une nouvelle version de NIO. Une nouvelle API pour gérer les dates (c'était vraiment trop simple avec les classes Date et Calendar). Une gestion standard des caches d'objets. Une gestion des unités et des quantités (JSR 275). Des apports à l'API de gestion de la concurrence, etc., etc.

Comme on ne peut pas parler de tous ces sujets – d'autant plus que beaucoup sont encore à l'état d'ébauches – je ne vais présenter en détail que deux des plus significatifs : les closures et les modules Java.

Les "closures"

Les "closures", fermetures en Français (aussi appelées "expressions lambda") ne sont pas une nouveauté dans les langages informatiques : des langages aussi divers que Smalltalk, JavaScript, C# ou Ruby en disposent déjà. Le tour de Java arrive. De quoi s'agit-il ? Il s'agit de définir, de manière légère, des fonctions capables de disposer du contexte d'un objet ou d'une autre fonction. Java dispose depuis longtemps, avec les classes anonymes, d'un mécanisme qui s'en rapproche. La version 7 propose donc une extension qui facilite encore davantage l'écriture de fonctions à la volée.



Une "closure" est un objet qui se définit de la façon suivante :

```
{int, int => int} fctAdd = {int x, int y => x+y};
```

La référence fctAdd désigne la closure. La déclaration {int, int => int} indique que fctAdd peut référencer une closure qui accepte en paramètre deux entiers et retourne un entier. L'objet closure est défini par {int x, int y => x+y}. Les paramètres en entrée sont nommés x et y. Leur somme est retournée par la closure.

La syntaxe d'une closure est la suivante :

```
{[parametres] => [instructions][valeur retournée]};
```

Le code contenu dans une closure peut être arbitrairement complexe. La closure suivante retourne la valeur absolue d'une valeur passée en paramètre. Elle intègre pour cela des instructions devant l'expression finale :

```
{int=>int} fctAbs = {int x=>if (x<0)x=-x};
```

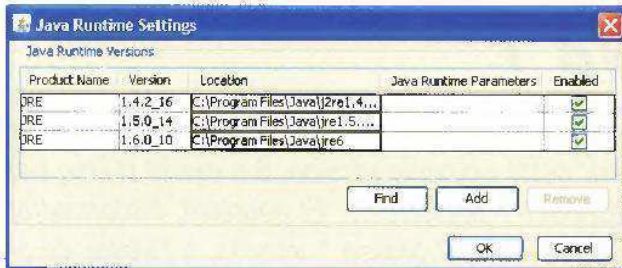
Ou le "x" qui suit le ";" est l'expression qui est retournée par la closure. Elle peut très bien n'avoir ni paramètre, ni valeur de retour :

```
{=>} fctHello = {=>System.out.println("Hello world");};
```

L'invocation d'une closure est réalisée par l'appel à la méthode "invoke" qui est définie pour toute closure. La signature de cette méthode invoke correspond aux paramètres et à la valeur de retour de la closure.

Ainsi on écrira :

```
int somme = fctAdd.invoke(3,4);
int valAbs = fctAbs.invoke(-5);
fctHello.invoke();
```



Le mécanisme prend toute sa saveur lorsque la closure utilise des attributs ou des variables (même locales !) accessibles à l'endroit où elle a été définie :

```
class Sprite {
    int x; int y;

    public void avancer() { agir({=>x+=1;}); }
    public void reculer() { agir({=>x-=1;}); }
    public void monter() { agir({=>y=1;}); }
    public void descendre() { agir({=>y+=1;}); }

    void agir({=>} action) {
        action.invoke();
        refreshDisplay();
    }
}
```

Cet exemple montre tout l'intérêt des closures : il devient possible de " passer " des instructions comme paramètres à une méthode. La chose était déjà possible en Java, mais de façon nettement plus lourde en implémentant à la volée l'interface Runnable :

```
public void avancer() {
    agir(new Runnable() {
        void run() { x +=1 ; }
    });
}

void agir(Runnable action) {
    action.run();
    refreshDisplay();
}
```

Bien sûr, l'utilisation de l'interface Runnable n'est possible ici que parce que la méthode d'action n'a besoin d'aucun paramètre, ni de valeur de retour. Si ce n'était pas le cas, il faudrait définir une interface ad hoc. C'est tout cela que nous épargne la closure.

Il est même légal d'écrire :

```
class Sprite {
    {int=>} fctAvancer;
    {int=>} fctReculer;

    public void initActions() {
```

```
int x = (int)(Math.random()*100);
fctAvancer = {=>x=x+1;x};
fctReculer = {=>x=x-1;x};
}
}
```

Surprenant, n'est ce pas ? Notre objet n'a plus d'attribut de positionnement. Celui-ci est défini comme variable locale d'une méthode et est ensuite manipulé par deux closures. Celles-ci peuvent être utilisées en dehors de la méthode initActions. Et pourtant la variable x est toujours disponible ! Cela n'est possible que parce que les variables locales manipulées par des closures sont allouées sur le tas (heap) et non sur la pile. Noter que nul spécificateur " final " n'est requis.

Le compilateur matérialise les closures en générant à la volée une interface et une classe implémentant cette interface. L'interface comme la classe ne définissent qu'une méthode : invoke, que nous avons déjà présentée.

Les closures sont bien sûr compatibles avec les autres mécanismes du langage comme les annotations ou les generics. Que vont-elles nous apporter ? Une nouvelle manière de programmer probablement. On va pouvoir, grâce à elles, accrocher facilement des comportements à des objets. C'est une façon très souple pour spécialiser le comportement d'un objet.

Les modules Java

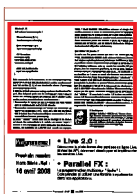
Les développeurs Java qui se sont un tant soi peu frottés à des applications d'entreprise ont nécessairement connu le " JAR Hell ", l'enfer des JAR : Les mêmes bibliothèques, utilisées par d'autres bibliothèques, dans des versions différentes, pas toujours compatibles ... Amis développeurs, votre souffrance a été entendue : SUN s'est décidé à introduire les modules Java sur sa plate-forme.

Un module Java est, comme un jar, une collection de classes compilées rassemblées dans un fichier zip, tout comme un fichier JAR. Il contient en plus une définition de module contenue dans un répertoire nommé MODULE-INF. Ce répertoire contient un fichier METADATA.module. Ce fichier contient des informations qui permettront à la JVM (Java Virtual Machine) de l'exploiter :

- Un nom de module.
- Sa version.
- La liste des modules dont il dépend avec éventuellement des contraintes sur les versions acceptables.
- La liste des points d'entrées publiques du module (les classes visibles en dehors du module).
- La classe d'amorçage du module (à l'instar de ce qui se fait dans les JAR). La fabrication d'un module Java est réalisée en écrivant un fichier " super package ". Il s'agit d'un fichier java (super-package .java) contenant les informations que l'on retrouvera dans le fichier METADATA.module. Ces informations seront aussi accessibles par introspection (une méthode getModule() sera disponible dans l'API de la classe ClassLoader). Un fichier super-package aura une structure qui devrait ressembler à cela (le conditionnel est de mise, la spécification n'a pas encore été validée) :

```
import java.module.annotation.* ;

@ModuleLayer("com.myapp.mypkg.Launch") ;
```



```
@Version("1.1")
superpackage com.myapp.mypkg {

    @VersionConstraint("2.1+")
    import com.myapp.mybasepkg;

    @VersionConstraint("1.2*")
    import com.myapp.myotherpkg;

    member package
    com.myapp.mypkg
    com.myapp.mypkg.service
    com myapp.mypkg.gui

    export com.myapp.Launch;
    export com.myapp.Service;
}
```

Cette structure de fichier est nouvelle. Le mot clé superpackage introduit le nom du module (com.myapp.mypkg). Il peut être annoté pour indiquer sa version (1.1) et une éventuelle classe d'amorçage (com.myapp.mypkg.Launch).

A l'intérieur de la structure superpackage, on trouve des directives d'import. Elles indiquent quels sont les modules dont notre module a besoin. Ces directives peuvent être annotées afin de préciser quelles sont les versions acceptables de ces modules externes :

- Une contrainte de version de type X.Y+ indique que toute version égale ou postérieure à la version X.Y est acceptée
- Une contrainte de version de type X.Y* indique que toute version de type X.Y est acceptée (X.Y, X.Y.1, X.Y.2, etc.), mais pas une version dont le numéro X ou Y est incrémenté (pour une contrainte 1.2, une version 2.0 ou 1.3 ne sera pas acceptée).

La clause " member " indique quels sont les packages et les classes qui devront être embarqués dans le module. Les clauses " export " dressent la liste des classes qui seront accessibles depuis les autres modules. Un mécanisme de référentiel sera associé aux superpac-

kages. Il devra permettre d'enregistrer, retrouver et charger des modules, et ce avec un niveau de contrôle très supérieur au mécanisme actuel, basé sur les JAR et le CLASSPATH, qu'il devrait donc remplacer. On se demande, en voyant tout ce qui est annoncé, comment la compatibilité ascendante pourra être assurée. A priori, les modules Java devraient être compatibles avec les versions inférieures du JDK (avec des fonctionnalités dégradées probablement).

Que faut-il en penser ?

Le moins que l'on puisse dire est que Java 7 est encore dans les limbes. Les promesses semblent aussi riches que lointaines. D'autant plus que pour en profiter pleinement, il faudra que les IDE comme Eclipse, NetBeans ou IdeaJ en tiennent compte (évolution du compilateur et de l'analyseur syntaxique, prise en compte des nouvelles API graphiques dans les composeurs graphiques, et ainsi de suite). Combien de temps faudra-t-il ? Plusieurs années sans doute ... Ce constat est un peu inquiétant car il montre un certain ralentissement dans l'évolution de la plate-forme Java. Est-ce la rançon de l'ouverture en " open source " ? Est-ce le destin d'une plate-forme déjà très complète qui repousse toujours plus loin les frontières du possible ? Est-ce une forme de prudence pour être davantage à l'écoute de la communauté des développeurs, dont le rôle décisif – démontré par les initiatives Hibernate ou Spring – ne peut plus être ignoré ? Il est aussi nécessaire que les développeurs Java " digèrent " les apports de la version 1.5 (et dans une moindre mesure 1.6). Nous avons dès maintenant les annotations et les generics. Mais leur utilisation reste encore assez limitée. La version 7 en tirera un meilleur profit. Combiné aux closures – une nouveauté de la version 7 – la programmation Java pourrait en être complètement bouleversée.



■ Henri Darnet

Directeur Technique - Objet Direct / Homsys Group
Objet Direct est une société spécialisée sur les technologies objet et Web : Conseil en méthodologie, en architecture et en urbanisation du SI, conception et mise en œuvre de projets, édition et distribution de logiciels, formation.
Objet Direct est une filiale de Homsys Group.
www.objetdirect.com