

# ClassLoader

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



***Préservez l'environnement,  
n'imprimez pas ce document***

## TABLE DES MATIERES

Java est paresseux.....	3
Si la classe n'est pas trouvée ? .....	3
Des espaces de noms .....	4
Isolation des applications WEB.....	4
Comment utiliser un class loader ? .....	5
Contourner la vérification des classes .....	6
Class loader et JSP.....	7
Class loader et RMI.....	7
EMail-let .....	8
Ecrire son class loader.....	8
Et les ressources ? .....	9

## Avant propos

Ce document passe en revue un élément important et innovant de Java par rapport aux autres langages : le chargeur de classes ou "class loader". Cet outil – en réalité une classe – s'occupe de rapatrier dans l'environnement d'exécution Java le code manquant et nécessaire.

## JAVA EST PARESSEU

Une des spécificités particulièrement innovante de Java est l'utilisation d'un class loader programmable. Une instance particulière s'occupe de charger dans la machine virtuelle les différentes classes au fur et à mesure des besoins. En effet, Java est un langage paresseux. Contrairement au C++ par exemple, Java ne charge en mémoire que les classes utiles à un moment donné. Lorsqu'une classe est nécessaire, elle est demandée à un class loader. Celui-ci doit charger et installer la classe en mémoire et retourner l'instance `Class` correspondante. Pour cela, des méthodes permettent de convertir un ensemble d'octets au format `.class`, en une classe. Le class loader doit fournir ces octets à la fonction `defineClass()`.

Les octets peuvent provenir de différentes sources. Le comportement par défaut consiste à trouver le fichier `.class` de la classe à installer dans des fichiers ou des archives au format JAR. La variable d'environnement `CLASSPATH` permet d'indiquer au class loader par défaut les différents endroits sur le disque où chercher les classes.

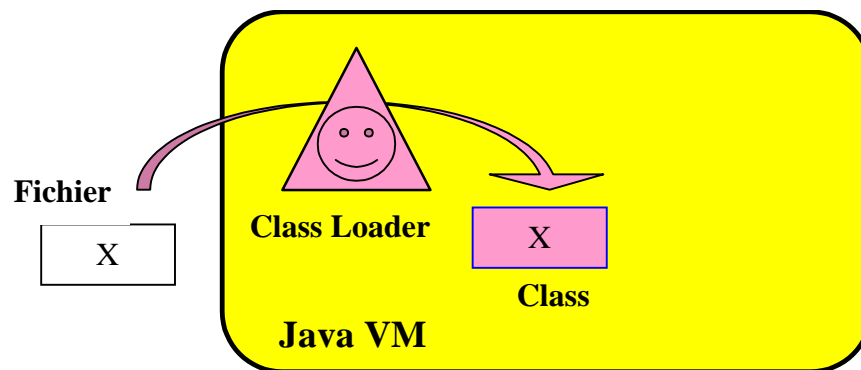


Figure 1 : Chargement en mémoire

Le class loader permet à la machine virtuelle de Java de n'avoir en mémoire que les classes utiles. Par exemple, si une application propose un service d'impression et que l'utilisateur ne l'invoque pas, les classes correspondantes ne seront pas chargées en mémoire. Par contre, lors de la première impression, la machine virtuelle devra invoquer régulièrement le class loader pour obtenir en mémoire l'ensemble des classes nécessaires à ce traitement.

Pour anticiper le chargement des classes, il faut écrire un traitement en tâche de fond avec une priorité basse, ayant pour charge de demander au class loader de charger certaines classes. Ainsi, lorsque le traitement principal en aura besoin, elles seront déjà présentes dans la JVM. La difficulté consiste à identifier les classes à charger et dans quel ordre. En effet, certaines classes peuvent avoir des effets de bords différents suivant l'ordre de chargement. Le chargement en tâche de fond doit respecter l'ordre naturel de chargement des classes.

Le class loader permet également d'obtenir les ressources présentes dans les archives grâce aux méthodes `getResource()` et `getResourceAsStream()`.

## SI LA CLASSE N'EST PAS TROUVEE ?

La recherche des fichiers `.class` dans les différents répertoires et dans les différentes archives prend du temps. Le class loader doit ouvrir toutes les archives et consulter les index pour trouver la classe. Pour optimiser cela, la version 1.3 des JVM permet d'indiquer dans un fichier d'index les différents packages disponibles dans l'archive. Le paramètre `-i` de l'utilitaire `jar` permet la construction d'un fichier `meta-inf/Index.list` avec la liste de tous les packages présents dans l'archive. Ce fichier est le premier de l'archive. Ainsi, le class loader peut connaître immédiatement les packages présents dans l'archive.

Pour détecter l'absence d'une classe, le class loader doit avoir consulté tous les répertoires et toutes les archives. Il est conseillé de réduire les chemins et les archives de la variable `CLASSPATH`.

Certaines API de Java utilisent l'absence de classe comme une information pertinente. Par exemple, le mécanisme de gestion des ressources en plusieurs langues utilise cela pour sélectionner le fichier ou la classe de ressources pour une langue donnée. La méthode `ResourceBundle.getResourceBundle("MaRessource", lang1)` va demander au class loader, successivement :

```
"MaRessource_" + lang1 + "_" + pays1 + "_" + variante1
"MaRessource_" + lang1 + "_" + pays1 + "_" + variante1 + ".properties"
"MaRessource_" + lang1 + "_" + pays1
"MaRessource_" + lang1 + "_" + pays1 + ".properties"
"MaRessource_" + lang1
"MaRessource_" + lang1 + ".properties"
"MaRessource_" + lang2 + "_" + pays1 + "_" + variante2
"MaRessource_" + lang2 + "_" + pays1 + "_" + variante2 + ".properties"
"MaRessource_" + lang2 + "_" + pays1
"MaRessource_" + lang2 + "_" + pays1 + ".properties"
"MaRessource_" + lang2
"MaRessource_" + lang2 + ".properties"
```

```
"MaRessource"
"MaRessource.properties"
```

La recherche s'effectue d'abord avec la langue demandée, puis avec la langue par défaut. Chaque échec entraîne la recherche de la version suivante. Cette approche sympathique pour l'utilisateur n'est pas très efficace pour le class loader. En effet, pour chaque ligne, il doit rechercher dans tous les répertoires et dans toutes les archives si la version est présente.

Le class loader peut être optimisé s'il garde en mémoire l'index de chaque archive. Le fichier d'index présent dans les archives 1.3 permet d'omettre les archives ne possédant pas de classe pour le package recherché.

## DES ESPACES DE NOMS

Le class loader permet de regrouper les classes dans un espace de nom. Deux classes de même nom peuvent être présentes dans la machine virtuelle à condition d'avoir été chargées par deux classes loaders différentes. C'est pour cela qu'il ne faut jamais comparer les classes de deux instances par leurs noms. Un des anciens bug de sécurité des premières machines virtuelles venait de là. Il était possible de prendre une instance chargée par un class loader et de la faire passer pour une instance chargée par un autre. Pour comparer les classes de deux instances, utilisez :

```
instance1.getClass() == instance2.getClass()
```

Même les classes systèmes comme `java.lang.String` peuvent être chargées par un class loader spécifique. Cela perturbe trop la machine virtuelle. Il est préférable de laisser le class loader system s'occuper des classes du package `java.lang`. Toutes les autres classes peuvent être chargées par votre propre class loader.

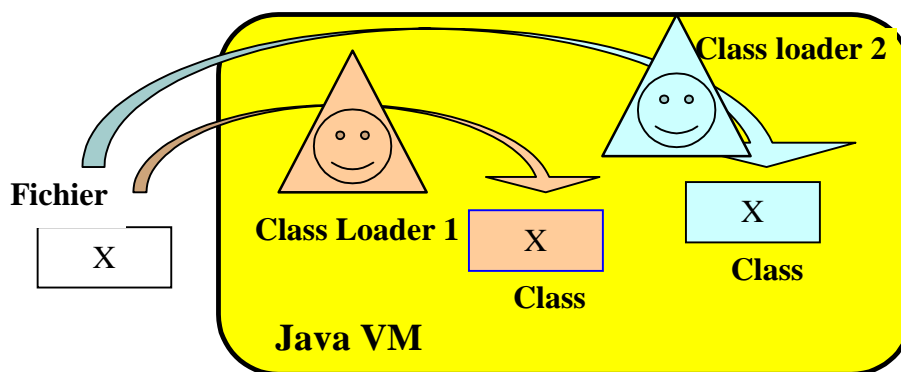


Figure 2 : Espaces de noms

Cette possibilité particulière est très intéressante. En effet, avec les langages classiques, il est très difficile d'intégrer des bibliothèques ayant des versions différentes. Java permet cela. C'est d'ailleurs recommandé dans les spécifications 2.3 des moteurs de servlet. Chaque application WEB doit être chargée par un class loader différent.

Cela permet, par exemple, à une application d'utiliser une version de l'analyseur XML différente de celle utilisée par le serveur d'application lui-même. Auparavant, il y avait des conflits entre les classes utilisées par l'application WEB et les classes utilisées par le serveur d'application. Maintenant, chacun est dans son espace de nom.

## ISOLATION DES APPLICATIONS WEB

Par contre, cela isole les différentes applications les unes des autres. C'est pour cela qu'il n'est plus possible de partager une session entre deux applications WEB. Analysons ce qu'il se passe lorsque deux applications désirent partager une instance. La première application demande à son class loader de charger la classe `MonObjet` à partir de l'archive `monobjet.jar`.

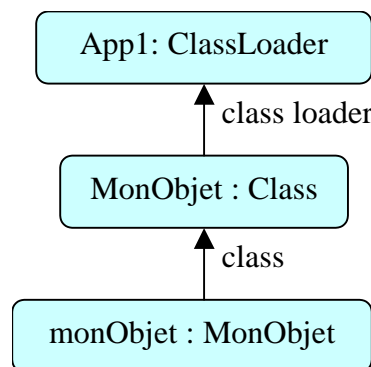


Figure 3 : Références

L'objet créé possède une référence sur sa classe et par contre coup sur le classloader de celle-ci.

L'application 2 obtient l'instance `monObjet` à partir de la session de l'utilisateur. Il obtient un pointeur de type `Object` sur l'instance.

```
Object obj=session.getAttribute("monObject");
```

Il désire alors convertir le pointeur en `MonObjet`.

```
MonObjet monObjet=(MonObjet)obj;
```

Du point de vue de l'application 2, **cette classe n'est pas encore en mémoire**. Le class loader de l'application 2 charge alors le fichier `MonObjet.class` de l'archive `monobjet.jar` et installe à nouveau la classe dans la machine virtuelle, mais cette fois-ci, associé au deuxième class loader.

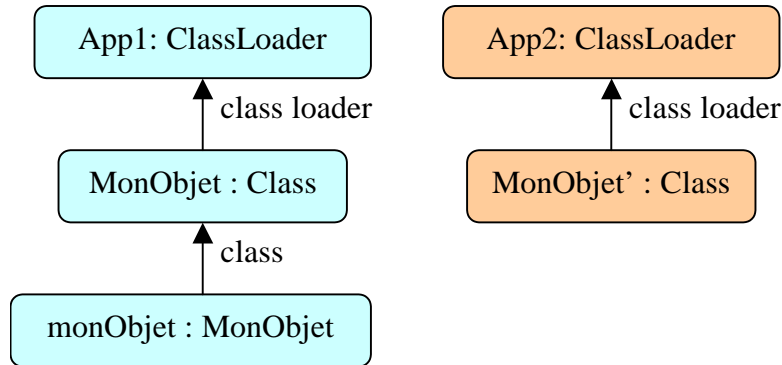


Figure 4 : Classes de même nom

La conversion du pointeur ne peut aboutir car l'instance n'est pas considérée de la même classe. Les noms des classes sont identiques mais les Class loaders sont différents. La JVM considère alors les classes comme différentes.

L'isolation des applications WEB dans des class loaders différents interdit le partage d'information entre deux applications WEB. Il faut modifier les anciennes applications WEB pour tenir compte de cela.

## COMMENT UTILISER UN CLASS LOADER ?

Pour isoler des classes dans un espace de nom, il faut les charger à partir d'un class loader et créer une première instance.

```
new MonClassLoader().loadClass("MaClass").newInstance();
```

Cela permet de demander la création d'une instance à partir d'un class loader spécifique. Lors du chargement de la classe `MaClass` par le class loader, toutes les classes utiles à celle-ci sont également chargées. Par exemple, la classe `MaClass` hérite de la classe `java.lang.Object`. La machine virtuelle demande alors au class loader d'installer celle-ci. En général, le class loader demande au class loader system s'il ne possède pas déjà une version. Si c'est le cas, la classe `java.lang.Object` n'est pas chargée à nouveau. Une référence sur la version installée par le classloader system est retournée.

Ce mécanisme est récursif. Si une classe périphérique a besoin d'une autre classe, elle est demandée au même class loader.

Vous vous dites : « mais alors, toutes les classes de l'application vont être chargées en mémoire lors de la demande de la première classe ! » Oui et non. En fait, certaines classes en relation avec la classe initiale seront chargées, d'autres non. Les classes et les interfaces héritées seront bien entendu demandées au class loader. Les classes des attributs également. Pour les classes utilisées dans les méthodes, cela dépend.

Dans le cas standard, les classes utilisées dans les méthodes ne sont pas demandées tant que la méthode n'est pas invoquée. En effet, elles ne sont pas nécessaires si les méthodes ne sont pas utilisées.

Java possède un algorithme de vérification du byte-code. Lorsque cet algorithme est utilisé, toutes les méthodes des classes sont analysées pour vérifier qu'il n'existe pas de byte-code qui viole les règles du langage. Pour faire cette analyse, il est parfois nécessaire de charger d'autres classes. Celles-ci ne sont pas installées dans la machine virtuelle. Leurs structures sont analysées pour vérifier que les traitements des classes appelantes sont correctement effectués. Par exemple, si une méthode d'une classe indique qu'elle retourne un entier, l'analyseur de byte code peut vérifier que le retour de l'invocation est bien considéré comme un entier par la méthode appelante et non comme un flotant ou une référence vers un objet. Les classes demandées par l'analyseur de byte code sont gardées dans un cache du class loader pour optimiser leurs installations futures dans la JVM.

Vous pouvez connaître les différentes classes chargées par la machine virtuelle en indiquant le paramètre `-verbose:class` lors du lancement de votre application. La machine virtuelle trace alors tous les chargements de classes. Par exemple, le lancement de `java -verbose:class java.lang.String` affiche la Figure 5.

```
[Opened c:\java\jdk1.3\jre\lib\rt.jar]
[Opened c:\java\jdk1.3\jre\lib\i18n.jar]
[Opened c:\java\jdk1.3\jre\lib\sunrsasign.jar]
[Loaded java.lang.Object from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.io.Serializable from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Comparable from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.String from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Class from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Cloneable from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.ClassLoader from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Throwable from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Error from c:\java\jdk1.3\jre\lib\rt.jar]
```

```
[Loaded java.lang.ThreadDeath from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.Exception from c:\java\jdk1.3\jre\lib\rt.jar]
[Loaded java.lang.RuntimeException from c:\java\jdk1.3\jre\lib\rt.jar]
...
```

Figure 5 : Traces obtenues

## CONTOURNER LA VERIFICATION DES CLASSES

Comprendre ce comportement peut permettre d'adapter le code pour des situations critiques. Par exemple, imaginons un code qui désire invoquer une méthode, si et seulement si, elle est présente dans la machine virtuelle. Cela peut arriver dans le cas d'un code devant s'adapter à différentes versions de bibliothèques. Par exemple, nous désirons répondre à la méthode `getServletInfo()` d'une servlet. Cette méthode doit retourner un descriptif de la servlet. Nous désirons retourner une chaîne de caractère du type « MaServlet v1.0 » ou le nom de l'application et le numéro de version sont obtenus à partir des informations indiquées dans l'archive `.jar` (Il est possible d'indiquer cela dans le fichier `meta-inf/Manifest.mf` de l'archive). Pour cela, nous devons invoquer la méthode `Class.getPackage().getImplementationVersion()`. Un code comme indiqué ci-après (source 1) semble suffisant.

```
public String getServletInfo()
{
    Package pack=getClass().getPackage();
    return pack.getImplementationTitle()+ ' '+
           pack.getImplementationVersion() ;
}
```

Cela fonctionne avec une JVM 1.2 ou de niveau supérieur. Mais, avec une machine virtuelle de niveau inférieur, la classe `java.lang.Package` n'existe pas. Le vérificateur de code va rejeter le chargement de la classe car il n'est pas capable de vérifier la méthode `getServletInfo()`.

Pour corriger cela, il faut utiliser une classe intermédiaire s'occupant d'invoquer le traitement.

```
class GetVer
{
    String getVer()
    {
        Package pack=getClass().getPackage();
        return pack.getImplementationTitle()+ ' '+
               pack.getImplementationVersion() ;
    }
}
public class MaServlet extends HttpServlet
{
    public String getServletInfo()
    {
        try
        {
            Class.forName("java.lang.Package"); // Check if JDK1.2
            return new GetVer().getVer();
        }
        catch (ClassNotFoundException x)
        {
            // Ignore
        }
        return "MaServlet";
    }
    ...
}
```

Ainsi, la classe `MaServlet` peut être vérifiée. Sa méthode `getServletInfo()` utilise la classe `GetVer` et respecte son utilisation. La classe `GetVer` est chargée par la machine virtuelle, mais elle n'est pas installée ni vérifiée. Le vérificateur de byte-code s'occupe uniquement des interfaces de cette classe. Si la construction de l'instance `GetVer` n'est jamais invoquée, la classe `GetVer` ne sera jamais installée et la classe `java.lang.Package` également.

Une approche raccourcie consiste à utiliser une classe interne (j'ai écrit un article à ce sujet disponible sur mon site [www.philippe.prados.net](http://www.philippe.prados.net)).

```
public class MaServlet extends HttpServlet
{
    public String getServletInfo()
    {
        try
        {
            Class.forName("java.lang.Package"); // Check if JDK1.2
            class GetVer {
                String getVer() throws ClassNotFoundException {
                    Package pack=getClass().getPackage();
                    return pack.getImplementationTitle()+ ' '+
                           pack.getImplementationVersion() ;
                }
            }
        }
    }
}
```

```

    }
    }
    return new GetVer().getVer();
}
}
catch (ClassNotFoundException x)
{
    // Ignore
}
return "MaServlet";
}
}

```

Nous avons ainsi contourné la vérification de byte-code à l'aide d'une classe intermédiaire. Cette approche est utile, par exemple, pour interfacer une application avec différents analyseurs XML. En effet, il n'existait pas de norme pour invoquer les analyseurs XML. Sun vient de remédier à cela en proposant les spécifications JAXP. Tous les analyseurs n'intègrent pas encore cette norme. Pour s'interfacer avec différents analyseurs il faut utiliser une classe intermédiaire pour chaque version. Ainsi, l'application peut détecter automatiquement l'analyseur présent dans la machine virtuelle et l'utiliser.

## CLASS LOADER ET JSP

Les pages JSP sont des pages HTML étendues, converties en fichier source java avant d'être compilées. On imagine alors très bien que le CLASSPATH doit pointer vers le répertoire temporaire où les fichiers JSP sont compilés. Ainsi, le class loader peut installer de manière paresseuse les nouvelles classes. Par exemple, lors de la demande de la page `toto.jsp`, le serveur d'application génère un fichier `toto_0.java`; lance le compilateur java afin d'obtenir le fichier `toto_0.class`; puis demande à un class loader d'installer cette classe.

Si une page JSP est modifiée, elle est automatiquement recompilée. Une nouvelle version de la classe est créée, avec comme suffixe le numéro de la nouvelle version, `toto_1.java` par exemple.

Cela peut être démontré en affichant la valeur d'une variable globale dans une page `test.jsp`.

```

<%! static int compteur=100; %>
<html>
<head>
<title>Test ClassLoader</title>
</head>
<body>
Valeur du compteur = <%= compteur++ %>
</body>
</html>

```

### Source 1

Lors de l'affichage de la page `test.jsp`, le navigateur indique :

```
Valeur du compteur = 100
```

En rafraîchissant la page, la valeur du compteur s'incrémente.

```
Valeur du compteur = 101
```

Si maintenant vous modifiez la page `test.jsp`, en ajoutant un espace par exemple, la page affichée reset le compteur à 100.

```
Valeur du compteur = 100
```

La classe `test_0` est toujours présente en mémoire. Elle ne peut pas être supprimée de la machine virtuelle pour plusieurs raisons : il est possible qu'une page de la version zéro soit en cours de traitement lors de la détection de la nouvelle version. Les traitements en cours doivent continuer. Au fur et à mesure, il n'y aura plus de traitement ni d'instance de la classe `test_0`. Ensuite, lorsqu'il n'existe plus d'instance `test_0`, la machine virtuelle ne peut pas savoir s'il n'est pas nécessaire d'en construire une nouvelle plus tard. De même, les variables globales de la classe `test_0` peuvent être référencées quelque part.

Les spécifications de java indiquent que les instances `Class` peuvent être supprimées de la mémoire lorsqu'il n'existe plus d'instance de ce type, et plus de référence vers le class loader ayant installé la classe. Le ramasse-miettes qui désire supprimer un class loader peut supprimer également les classes associées. Il est certain qu'il ne sera plus possible d'utiliser les classes. Il faut donc que toutes les instances créées par ce class loader soient détruites. La méthode statique `classFinalize()` est invoquée pour chaque classe avant destruction.

Pour nettoyer les différentes versions des pages JSP du serveur d'application, il faut perdre la référence sur le class loader correspondant. Suivant les implémentations, arrêter l'application et la relancer peut entraîner le nettoyage des classes. Si le moteur J2EE supprime le class-loader de l'application lors de son arrêt, toutes les classes associées seront nettoyées à terme.

## CLASS LOADER ET RMI

Java propose un ORB (Object Request Broker) pour permettre l'invoque de méthodes sur des objets présents sur une autre machine : RMI (Remote Methode Invocation). Le protocole technique de la couche réseau est IIOP, le même qu'utilise les ORB CORBA. Les objets distants sont vus comme des objets locaux. Imaginons la situation suivante : un objet `Client` invoque la méthode `getUtilisateur()` d'un objet `Serveur` distant. La méthode `getUtilisateur()` retourne une instance `Utilisateur`, inconnu par le class loader du client.

Comment construire une instance `Utilisateur` sur le client ? L'ORB RMI possède un class loader particulier qui va se charger de récupérer les classes absentes sur le poste du client, à partir du serveur. Pour construire l'instance `Utilisateur` retournée par la méthode

`getUtilisateur()` de l'instance sur le serveur, le class loader va interroger celui-ci pour lui demander le fichier `.class` correspondant. Le class loader RMI peut alors installer la classe dans sa JVM et construire l'instance à partir des informations du protocole IIOP.

Le paramètre d'environnement `java.rmi.server.codebase` permet d'indiquer les différents répertoires et les différentes archives permettant au client de télécharger les classes du serveur.

```
java -Djava.rmi.server.codebase=http://www.philippe.prados.net/export/ ...
```

Avec ce mécanisme, le CLASSPATH du client peut être très léger. Seules les classes du client sont nécessaires. Si les classes du serveur évoluent, il n'est pas nécessaire de mettre à jour chaque client. Le class loader se charge d'obtenir les nouvelles versions du serveur.

Sans ce mécanisme, si deux versions différentes de la même classes sont présentes sur le client et le serveur, le client ne sera pas capable de construire l'instance `Utilisateur`. Le numéro de version de la sérialisation ne serait pas correct.

Attention, cela entraîne également qu'il est possible d'exécuter sur le poste du client des classes non connues par celui-ci. Un pirate qui arriverait à modifier le serveur peut propager un virus sur tous les clients ! Il est nécessaire d'encadrer les traitements client d'un class loader sécurisé. Le class loader sécurisé associe un signataire à une classe. Ainsi, les API peuvent autoriser ou interdire certaines fonctions. Interrogez vos fournisseurs de moteur J2EE pour connaître les protections mises en place contre cela.

## EMAIL-LET

Les applets sont des petites applications chargées par les navigateurs dans une page. Un class loader s'occupe d'obtenir l'archive et les fichiers `.class` à partir du serveur ayant livré la page HTML.

Il est possible d'offrir un mécanisme similaire dans un email. Imaginez la situation suivante : vous construisez une archive avec les classes de votre application. Vous placez cette archive et une instance sérialisée décrivant l'invocation d'un traitement dans un email. Le message est envoyé dans la boîte aux lettres d'un serveur. Lorsqu'il le désire, le serveur consulte sa boîte aux lettres. Il extrait l'archive, branche un class loader sur celle-ci, et invoque le traitement décrit dans l'instance sérialisée. Le serveur retourne le résultat dans un message en réponse.

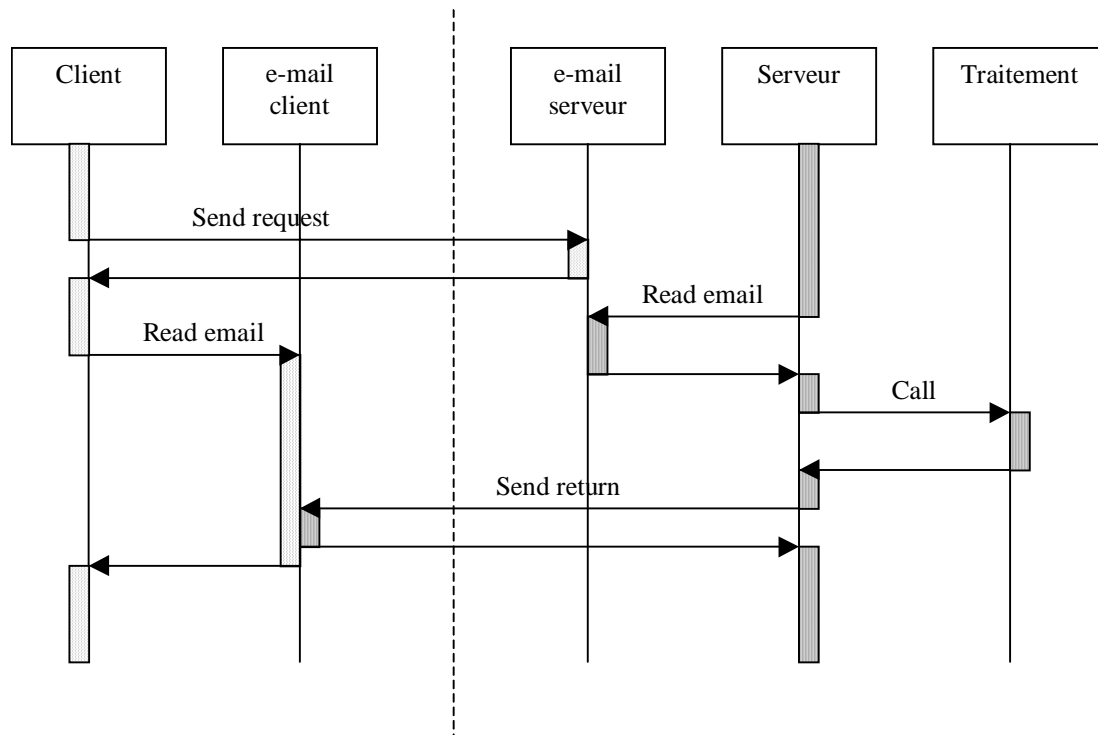


Figure 6 : Diagramme de séquence

En quelques heures, vous avez réécrit Message-Queue ou JMS ! Vous pouvez invoquer un traitement asynchrone. J'ai rédigé un code comme celui-ci, mais il est interne à IBM et ne peut être publié.

## ECRIRE SON CLASS LOADER

Nous allons écrire un class loader pour obtenir les fichiers `.class` et les ressources à partir du répertoire `/myclassloader/`. Le code peut être facilement adapté pour obtenir les classes à partir d'une base de donnée, d'une connexion réseau particulière, d'un email, etc.

Le JDK 1.2 simplifie la rédaction d'un class loader. Il suffit de surcharger la méthode `findClass()` pour charger une nouvelle classe. La classe `MyClassLoader`, Source 2, invoque la méthode `loadClassData()` pour obtenir un tableau de byte avec la description de la classe. Ce tableau est strictement le contenu d'un fichier `.class`. Dans cet exemple, le fichier `.class` est récupéré du répertoire `/myclassloader/`. Il est facile d'adapter cette méthode pour obtenir cette information à partir d'autres sources.

```
package net.prados.philippe.myclassloader;
public class MyClassLoader extends ClassLoader
```

```

{
    protected MyClassLoader()
    {
        super();
    }

    protected MyClassLoader(ClassLoader parent)
    {
        super(parent);
    }

    protected Class findClass(String name) throws ClassNotFoundException
    {
        byte[] b = loadClassData(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassData(String name)
        throws ClassNotFoundException
    {
        try
        {
            String filename="/myclassloader/"+name.replace('.', '/')+".class";
            InputStream in=new FileInputStream(filename);
            byte[] data=new byte[in.available()];
            in.read(data);
            in.close();
            return data;
        }
        catch (FileNotFoundException x)
        {
            throw new ClassNotFoundException(name,x);
        }
        catch (IOException x)
        {
            throw new ClassNotFoundException(name,x);
        }
    }
}

```

#### Source 2

La méthode `findClass()` est invoquée par la méthode `loadClass()` si la classe à charger n'a pas été trouvée dans les `ClassLoader` précédant ou système. Si vous désirez avoir une trace de toutes les classes chargées par le class loader, surchargez cette méthode comme indiqué ci-dessous :

```

protected Class loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
    System.err.println("[loadClass("+name+", "+resolve+"]");
    return super.loadClass(name, resolve);
}

```

#### Source 3

### ET LES RESSOURCES ?

Les classes-loaders ne s'occupent pas uniquement de charger les classes. Ils doivent également s'occuper de charger les ressources à partir des archives ou des autres technologies utilisées par les class loaders. Pour cela, il faut surcharger la méthode `findResource()`. Celle-ci doit retourner une URL.

Dans l'exemple qui nous concerne, il suffit de retourner une URL de type `file:` (Source 4).

```

protected URL findResource(String name)
{
    try
    {
        String filename="file:///myclassloader"+name;
        return new URL(filename);
    }
    catch (MalformedURLException x)
    {
        return null;
    }
}

```

**Source 4**

Souvent, il n'est pas possible d'utiliser les protocoles standards pour référencer une ressource. Par exemple, si la ressource est présente dans une archive, il faut retourner une URL particulière. Les protocoles standards ne savent pas référencer un fichier dans une archive. Il n'est pas possible d'utiliser `file:` ou `http:`. Nous allons alors ajouter un nouveau type de protocole, spécialisé dans le référencement des ressources livrées par le class loader. La méthode `findResource()` pourra alors retourner une URL spéciale, du type « `myclassloader://monrepertoire/monfichier.txt` ».

Pour faire cela, il faut ajouter dans la variable d'environnement `java.protocol.handler.pkgs` le nom d'un package. Le nom du protocole est ajouté à ce package, et une classe `Handler` est recherchée. Cette classe doit hériter de `URLConnectionHandler`.

Par exemple, nous avons utilisé le package `net.prados.philippe.myclassloader` pour notre classe loader. Nous allons ajouter le package `net.prados.philippe` dans la variable d'environnement (ci-dessous).

```
package net.prados.philippe.myclassloader;

class MyClassLoader
{
    ...
    static
    {
        Properties properties= System.getProperties();
        String s= properties.getProperty("java.protocol.handler.pkgs");
        if (s == null)
            s= "net.prados.philippe";
        else
            s= s + "|net.prados.philippe";
        properties.put("java.protocol.handler.pkgs", s);
    }
}
```

**Source 5**

Cette fonction statique est appelée lors du chargement de la classe du class loader. Il faut ensuite déclarer une classe `Handler` dans ce package.

```
package net.prados.philippe.myclassloader;

import java.io.*;
import java.net.*;
import java.util.*;

public class Handler extends URLStreamHandler
{
    public Handler()
    {
    }

    public URLConnection openConnection(URL url)
    {
        return new MyClassLoaderURLConnection(url);
    }
}
```

**Source 6**

Cette classe sera chargée lors de l'ouverture d'une URL dont le protocole est `myclassloader`. La classe `MyClassLoaderURLConnection` doit offrir une connexion vers la ressource (Source 7).

```
package net.prados.philippe.myclassloader;

import java.io.*;
import java.net.*;

public class MyClassLoaderURLConnection extends URLConnection
{
    protected MyClassLoaderURLConnection(java.net.URL url)
    {
    }

    public void connect() throws java.io.IOException
    {
    }

    public InputStream getInputStream() throws IOException
    {
    }
}
```

```
        return new FileInputStream(getURL().getFile());  
    }  
}
```

#### Source 7

La méthode `getInputStream()` doit se charger d'ouvrir un flux vers la ressource. Dans notre cas, la situation est simple, nous ouvrons simplement un fichier.

Avec tous cela, nous pouvons manipuler notre class loader pour trouver des classes et des ressources.

```
ClassLoader cl=new MyClassLoader();  
cl.loadClass("MaClass").newInstance();  
cl.getResource("/coucou.txt");
```