Chargeurs de classe

Université de Nice - Sophia Antipolis
Richard Grin
Version 1.2.1 – 14/6/05

Introduction

Rôle des chargeurs de classes

- Java permet le chargement dynamique de classes pendant l'exécution d'un programme
- Les chargeurs de classes ont la tâche de charger les classes (et les ressources liées aux classes)
- Ce sont des objets Java ordinaires que le programmeur peut manipuler
- Ce cours décrit les chargeurs de classes (en Java 2) et montre comment en écrire de nouveaux
- Le cours sur la sécurité décrit leur rôle très important dans le modèle de sécurité de Java

Chargement des classes

- Au contraire d'autres langages comme le langage C, Java ne charge en mémoire les modules exécutables que lorsqu'ils sont vraiment nécessaires à l'exécution
- Des classes peuvent donc être chargées automatiquement par la JVM en cours d'exécution

Chargement implicite ou explicite des classes

- Le plus souvent les classes sont chargées car le code nécessite implicitement leur chargement ; par exemple, s'il faut exécuter une méthode d'une classe non encore chargée
- Une commande explicite peut aussi charger une classe dont on donne le nom

Chargement « tardif »

- Les classes ne sont chargées que lorsque la JVM en a vraiment besoin
- Ainsi, la déclaration d'une variable d'une classe non encore chargée ne provoque pas le chargement de la classe
- En anglais : « *lazy loading* » (mot à mot : « chargement paresseux »)

Classe **`Class`**

Classe **`Class`**

- Il est indispensable de bien connaître la classe **`Class`** pour étudier les chargeurs de classe
- Cette classe représente une classe Java chargée en mémoire pendant l'exécution d'un code Java
- Tout classe chargée dans la JVM pendant l'exécution est représentée par une instance de la classe **`Class`**

Classe **`Class`**

- A tout objet Java est associé une instance de la classe **`Class`** qui représente la classe de l'objet
- On obtient cette instance par la méthode **`Class getClass()`** de la classe **`Object`**
- Cette instance est créée automatiquement quand survient un des événements suivants :
  - un premier objet de cette classe est créé
  - une méthode **`static`** de la classe est appelée

Classe **`Class`**

- En fait, la création de l'instance de **`Class`** fait partie du processus du chargement de la classe effectué par un chargeur de classe
- Elle est effectuée par la méthode **`defineClass`** du chargeur de classe

Chargement explicite d'une classe

- Dans certaines occasions il est préférable, ou même nécessaire (comme pour le chargement des drivers JDBC) de faire charger explicitement une classe
- 2 façons utilisent le chargeur de classes qui a déjà chargé la classe qui contient le code explicite de chargement
- On peut aussi utiliser un autre chargeur de classes

Chargement explicite d'une classe

- Voici les 2 premières façons :
  - ```
    Class c =
      fr.unice.p1.UneClasse.class;
    ```
  - la 2ème façon est utilisable même si on ne connaît le nom de la classe qu'à l'exécution :

```
String nomClasse;
. . .
Class.forName(nomClasse);
```
```

Les méthodes forName (1)

- La classe **`Class`** possède 2 méthodes **`static forName`**
- 1. **`Class forName(String nomClasse)`** charge une classe dont on donne le nom complet (avec le nom du paquetage)
- Le chargement est effectué par la méthode **`loadClass`** du chargeur de classe de la classe qui contient l'appel à **`forName`**
- La classe est initialisée après son chargement (les blocs **`static`** sont exécutés et les variables **`static`** sont initialisées)

Les méthodes forName (2)

- 2. **`Class forName(String nomClasse, boolean initialise, ClassLoader cl)`** fait appel au chargeur **`cl`** pour charger une classe dont on donne le nom complet ; la classe est initialisée ou pas suivant la valeur du paramètre **`initialise`**
- Utilisation rare de **`initialise`** : repousser à plus tard l'exécution d'un bloc **`static`** lourd
- Si la classe n'est pas initialisée, elle le sera plus tard automatiquement avant son utilisation

Les méthodes forName (3)

- Remarque :
  `Class c =`
  `fr.unice.p1.UneClasse.class;`
  fait appel à la méthode **`forName`** à 1 paramètre

ClassNotFoundException

- Les méthodes **`forName`** (et **`ClassLoader.loadClass`**) peuvent renvoyer l'exception contrôlée **`ClassNotFoundException`**

Les erreurs

- Dans certains cas, le chargement d'une classe peut provoquer une **`NoClassDefFoundError`**
- Par exemple, lorsque un **`URLClassLoader`** récupère une classe dont le nom ne correspond pas à son emplacement dans le système de fichiers
- Autres erreurs possibles : **`ClassFormatError`**, **`VerifyError`**

Chargeur d'une classe

- Les classes conservent une référence au chargeur de classes qui les a chargées
- On peut récupérer ce chargeur par la méthode **`public ClassLoader getClassLoader()`** de la classe **`Class`** (qui renvoie **`null`** pour le chargeur interne)
- Réciproquement, les chargeurs de classes habituels conservent des références vers toutes les classes qu'ils ont chargées

Espaces de noms

- Le nom complet d'une classe comprend le nom du paquetage ; par exemple, **`java.lang.Object`**
- Cependant, 2 classes qui ont exactement le même nom complet, mais qui ont été chargées par 2 chargeurs de classes différents, sont considérées comme différentes par la JVM
- Les navigateurs utilisent cette propriété pour protéger les applets en créant un chargeur de classes par *codebase*

Mécanisme pour les espaces de noms

- Si une classe C a besoin du chargement d'une autre classe D pour fonctionner, le chargement de D est demandée au chargeur de C
- Si un autre chargeur, sans parenté avec le chargeur de C, a déjà chargé une classe D de même nom, elle ne sera pas connue du chargeur de C et la classe sera à nouveau chargée et sera considérée par la JVM comme différente de la classe chargée par l'autre chargeur

Les différents chargeurs de classes

Chargeur de classes interne à la JVM

- La JVM intègre un chargeur de classes, dit interne (ou principal, ou primordial)
- Il charge les classes de base de l'API (celles fournies avec le SDK) contenues dans le fichier **`rt.jar`** placé dans le répertoire d'installation du runtime de Java
- Ce chargeur de classes n'est pas représenté par une classe Java, il est écrit « en dur » dans la JVM

Autres chargeurs de classes

- Tous les autres chargeurs de classes sont représentés par une classe Java, sous-classe de la classe abstraite **`java.lang.ClassLoader`**
- Depuis Java 1.2, tout chargeur de classe a un chargeur parent qui est donné au moment de la création du chargeur
- On peut retrouver le parent par la méthode **`ClassLoader getParent()`** de la classe **`ClassLoader`** (renvoie **`null`** si le parent est le chargeur interne)

Recherche des classes

- Chaque chargeur de classes ont son propre mode de recherche des classes ; par exemple on peut écrire un chargeur qui recherche le *bytecode* des classes dans une base de données
- La plupart suivent un modèle de délégation pour rechercher une classe (depuis le SDK 1.2) : ils commencent par déléguer la recherche à leur parent, et ne recherchent eux-mêmes une classe que si leur parent ne l'a pas trouvé

Utilisation de la délégation

- Protection des classes système car elles ne peuvent être rechargées par un autre chargeur que le chargeur interne
- Souplesse pour le partage de classes : des classes peuvent être partagées entre plusieurs « domaines » associés à des chargeurs différents (si elles sont chargées par un chargeur parent)
- Si une interface est chargée par un chargeur parent, les classes qui implémentent cette interface peuvent être utilisées même si elles ont été chargées par un chargeur « inconnu »

Chargeurs de classes de base

- 2 autres chargeurs de classes de base sont fournis avec le SDK :
  - chargeur d'extensions qui chargent les classes d'extension placées dans le répertoire **`lib/ext`** placé dans le répertoire d'installation du *runtime* Java ; il a pour parent le chargeur interne
  - chargeur système (ou d'application) qui charge les classes depuis le *classpath*, en tenant compte du nom complet des classes ; il a pour parent le chargeur d'extensions

URLClassLoader

- Une instance de cette classe est un chargeur qui recherche des classes dont on donne l'URL
- Le *bytecode* de la classe cherchée peut être un simple fichier ou une entrée d'un jar
- La souplesse de ce chargeur permet souvent d'éviter d'avoir à écrire son propre type de chargeur
- Si malgré tout aucune des classes existante ne lui convient, le programmeur peut créer ses propres classes de chargeurs

Autres chargeurs de classes

- Tous les navigateurs ont leur propre chargeur de classes pour charger les applets
- **`java.rmi.server.RMIClassLoader`** permet de charger dynamiquement les classes quand on utilise RMI (n'hérite pas de **`ClassLoader`**)
  - il utilise le protocole HTTP pour charger les classes
  - la propriété **`java.rmi.server.codebase`** lui indique où trouver les classes

Utilisation d'un chargeur de classes

`loadClass`

- **`public Class loadClass(String nomClasse)`** (de la classe **`ClassLoader`**) permet de charger une classe (on donne le nom complet de la classe)
- Ensuite, on peut utiliser la méthode **`newInstance()`** de la classe **`Class`** pour créer une instance de la classe chargée (seulement possible si la classe possède un constructeur sans paramètre)
- On peut utiliser l'API de réflexivité **`java.lang.reflect`** pour appeler d'autres constructeurs

Les classes de l'API Java liées aux chargeurs de classes

Classe **`ClassLoader`**

- C'est une classe abstraite de **`java.lang`**, racine de tous les chargeurs de classes
- Les méthodes **`public`** :
  - **`Class loadClass(String nomClasse)`** Le nom de la classe est un nom complet, avec le nom du paquetage
  - des méthodes pour récupérer des ressources (vues précédemment)

Classe **`SecureClassLoader`** (1)

- Si on veut créer son propre chargeur de classes, il faut hériter de **`SecureClassLoader`** et pas de **`ClassLoader`**
- En effet, cette classe associe un *codesource* aux classes chargées et permet ainsi d'utiliser la notion de permission
- Ce codeSource indique d'où vient la classe et contient les certificats associés à la classe si elle est signée (voir cours sur la sécurité)

Classe **`SecureClassLoader`** (2)

- C'est la classe dont il faut hériter si on veut utiliser les nouvelles possibilités introduites en Java 2 pour choisir les domaines de sécurité associés aux classes
- C'est une classe fille de **`ClassLoader`**
- Elle est dans le paquetage **`java.security`**

`URLClassLoader`

- Cette classe de **`java.net`** peut charger des classes situées dans des répertoires ou des fichiers jar, sur la machine locale ou sur une machine distante
- C'est une instance de cette classe qui est utilisé pour charger les classes situées dans le *classpath* (en JDK 1.1, les classes du *classpath* était chargées par le chargeur interne)
- Cette classe suffit bien souvent au programmeur et lui évite de créer son propre chargeur de classes
- Elle hérite de **`SecureClassLoader`**

Créer un **`URLClassLoader`**

- 3 constructeurs (**à éviter** ; voir transparent suivant) :
  - **`URLClassLoader(URL[] urls)`** ; le parent sera le chargeur de
  - **`URLClassLoader(URL[] urls, ClassLoader parent)`**
  - **`URLClassLoader(URL[] urls, ClassLoader parent, URLStreamHandlerFactory factory)`**
- Le parent du chargeur construit avec le premier constructeur sera le chargeur système (celui qui cherche dans le classpath)
- Si on passe **`null`** comme 2ème paramètre du 2ème constructeur, le parent du chargeur construit sera le chargeur interne (qui charge les classes de l'API)

Créer un **`URLClassLoader`**

- En fait, il vaut mieux obtenir une instance par les méthodes de classe suivantes **`URLClassLoader newInstance(URL[] urls)`**
  - **`URLClassLoader newInstance(URL[] urls, ClassLoader parent)`**
- En effet, ces méthodes « fabriques » renvoient une instance qui testera les autorisations d'accès et de définition pour le paquetage des classes à charger (voir cours sur la sécurité en Java)

Types d'URL pour URLClassLoader

- URL d'un répertoire local ; attention, l'URL doit se terminer par un « / » ; l'oubli de ce « / » peut occasionner des erreurs difficile à trouver ; si on utilise file comme protocole, ne pas le faire suivre de « // » car il sera alors utilisé un socket et des permissions seront alors nécessaires (voir cours sur la sécurité)
- URL d'un fichier jar local : le plus simple est de passer par la classe **`File`** et d'utiliser la méthode **`toURL()`** (voir exemple plus loin)

Position des classes par rapport aux URL

- L'URL indique la racine pour trouver la classe ; l'endroit exact est calculé d'après le nom du paquetage de la classe à charger, aussi bien sous les répertoires que dans les fichiers jar

Exemple avec **`URLClassLoader`**

```
URL[] urls = {
  new URL("file:/u/toto/rep1/"),
  new URL("http://clio.unice.fr/~toto/cl.jar"),
  new URL("file:rep/classes/"),
  new File("cl.jar").toURL()
};
URLClassLoader ucl =
  URLClassLoader.newInstance(urls);
Class c = ucl.loadClass("fr.truc.Classe1");
Classe1 ic1 = c.newInstance();
```


Chargement d'une ressource

URL d'une ressource

- Les ressources liées à une classe (fichiers de propriétés, textes, images, sons,…) peuvent être désignées par un URL absolu, ou relatif ("/" comme séparateur)
  - absolu : la recherche est effectuée par le chargeur de classes avec l'URL inchangé (en général, à partir des entrées du classpath)
  - relatif : le chemin associé au paquetage de la classe est ajouté au début ; par exemple /fr/unice/toto/p1/ si la classe est du paquetage fr.unice.toto.p1, et ensuite la recherche se fait comme avec un chemin absolu

Ressources liées à une classe

- Pour récupérer une ressource utilisée par une classe, on demande à la classe de le faire avec une des méthodes **`getResource`** ou **`getResourceAsStream`**
- Elle délèguera la tâche au chargeur de classe qui l'a chargée (avec les méthodes de même nom de la classe **`ClassLoader`**)
- Mais avant de déléguer elle va faciliter la recherche des ressources placées relativement à son fichier **`.class`** (si on lui passe un nom relatif)

Méthodes pour charger les ressources

- On utilise une des 2 méthodes suivantes de la classe **`Class`** :
  - **`URL getResource(String nom)`** renvoie un URL de la ressource, qui peut être utilisé pour récupérer la ressource
  - **`InputStream getResourceAsStream(String nom)`** permet de récupérer en une seule étape un flot pour lire la ressource

Chargeur et ressources liées à une classe

- Les méthodes du transparent précédent donnent la 1ère ressource qui correspond au nom
- La méthode suivante de la classe **`ClassLoader`** donne tous les URL qui correspondent au nom **`Enumeration getResources(String nom)`**

Étapes de la récupération d'une ressource

- Les ressources sont recherchées en suivant les mêmes étapes que la recherche des classes
- La recherche est effectuée par le chargeur de classe qui a chargé la classe qui fait la recherche de la ressource
- Il essaie de déléguer la recherche à son chargeur père
- Normalement la recherche commence donc par les répertoires et fichiers (jar et zip) du *classpath*

Recherche dans le *classpath*

- La classe **`ClassLoader`** possède 3 méthodes **`static`** qui font rechercher les ressources par le chargeur de classe système (donc dans le *classpath*) :
  - **`URL getSystemResource(String nom)`**
  - **`InputStream getSystemResourceAsStream(String nom)`**
  - **`Enumeration getSystemResources(String nom)`**

Écrire un chargeur de classes

Conventions de base

- Le programmeur doit en principe respecter certaines conventions pour l'écriture de ces classes :
  - Hériter de **`SecureClassLoader`**
  - Déléguer la recherche des classes à son chargeur parent, avant de les chercher lui-même ; en particulier, le chargeur de classes interne doit rechercher les classes avant tous les autres chargeurs

Constructeur

- Le constructeur d'un chargeur de classes peut indiquer le chargeur parent de l'instance créée
- La classe **`SecureClassLoader`** a un constructeur sans paramètre et un autre qui prend le chargeur parent en paramètre ; il en est de même pour la classe **`ClassLoader`**
- Par défaut, le chargeur de classe parent est le chargeur « système » (appelé aussi application)

Méthode **`loadClass`** (1)

- Cette méthode effectue le chargement des classes
- En-tête de la méthode :

```
public Class loadClass(String nom,
                       boolean resolve)
  throws ClassNotFoundException
```

- Le 2ème paramètre indique si on doit charger tout de suite les classes dont dépend la classe
- Une variante de **`loadClass`** n'a que le nom de la classe à charger comme paramètre ; elle « résoud » la classe chargée

Méthode **`loadClass`** (2)

- Dans les versions antérieures à la version 1.2 elle était abstraite dans la classe **`ClassLoader`**
- On devait donc la redéfinir dans les classes filles pour écrire un nouveau type de chargeur de classes
- A partir de la version 1.2, la méthode **`loadClass`** n'est plus abstraite
- La classe **`ClassLoader`** n'a plus aucune méthode abstraite mais elle est tout de même abstraite ; on ne peut pas créer d'instance

Méthode **`loadClass`** (Java 1.2)

- Elle fait appel à la méthode **`protected Class findClass(String nomClasse) throws ClassNotFoundException`** dont l'implémentation par défaut lance une **`ClassNotFoundException`**
- On doit donc redéfinir cette méthode **`findClass`**
- Cette méthode doit rechercher le *bytecode* de la classe à charger et le « définir » (appel de la méthode **`define`**)
- Le reste du code de la méthode **`loadClass`** effectue tout le travail le plus complexe

Méthode **`loadClass`** (2)

- La méthode **`loadClass`** assure un comportement standard pour tous les chargeurs de classes qui ne redéfinissent que **`findClass`**
- Si ce comportement standard ne convient pas on peut toujours redéfinir la méthode **`loadClass`**
- Mais ceci doit être réservé aux programmeurs experts qui savent bien ce qu'ils font car un chargeur de classes est une brique essentielle du système de sécurité de Java

- Les transparents suivants décrivent le comportement standard de la méthode **`loadClass`**
- Ils donnent en fait le code de la méthode **`loadClass`** de la classe **`SecureClassLoader`**, héritée de **`ClassLoader`**
- Si on redéfinit la méthode **`loadClass`** on ne doit pas trop s'éloigner de ce comportement standard, par exemple pour déléguer à un autre chargeur de classes que le chargeur parent

Étapes de l'exécution de `loadClass`

1. Demande à la JVM si ce chargeur a déjà chargé la classe ; la renvoie si c'est le cas
2. Demande au chargeur parent (ou au chargeur interne) de charger la classe s'il le peut (délégation)
3. Recherche et lit le *bytecode* de la classe
4. « Définit » la classe : génère une instance de **`Class`** à partir du *bytecode* (avec intervention du vérificateur de *bytecode*)
5. Résout les références à d'autres classes (les charge s'il le faut) : classe mère,…

Étapes liées au paquetage

- On peut ajouter 2 étapes qui vérifient que l'on a bien les autorisations d'accès au paquetage de la classe à charger, et de créer une nouvelle classe dans ce paquetage :
  - avec les 2 méthodes **`checkPakageAccess`** et **`checkPakageDefinition`** de la classe **`SecurityManager`**

Sécurité et `loadClass`

- Les étapes de l'exécution de **`loadClass`** permettent d'éviter la création de classes qui remplacent les classes de l'API : ces classes sont chargées au début et seront donc récupérées à l'étape 2
- L'auteur d'un chargeur de classes doit être de confiance, respecter les étapes données ci-avant, et éviter si possible de redéfinir la méthode **`loadClass`** : il doit définir la méthode **`findClass`**

Schéma du code de loadClass (1)

```
// Si la classe a déjà été chargée par le même
// chargeur, on la renvoie
Class c = findLoadedClass(nom);
if (c == null) {
  // sinon on la recherche ;
  // code dans le transparent suivant
  . . .
}
// Résolution de la classe si demandé
if (resolve)
  resolveClass(c);
return c;
```


Schéma du code de loadClass (2)

```
try {
  if (parent != null) {
    c = parent.loadClass(nom, false);
  } else {
    // le parent est le chargeur système
    c = findBootstrapClass0(nom);
  }
} catch (ClassNotFoundException e) {
  // Ce chargeur cherche lui-même la classe
  c = findClass(name);
}
```

Écriture de **`findClass`**

- Cette méthode doit
  - trouver la source qui contient le *bytecode* (url, élément d'une base de donnée,…)
  - lire le *bytecode*
  - « définir » la classe, c'est-à-dire transformer les octets du *bytecode* en instance de la classe **`Class`** ; il suffit d'appeler la méthode **`defineClass`** définie dans la classe **`SecureClassLoader`**
- Elle n'a rien d'autre à faire puisque **`loadClass`** se charge du reste

Exemple schématique de code pour **`findClass`**

```
// Utilise le nom de la classe pour savoir où
// et comment aller lire les octets de la classe
// Par exemple, fr.unice.C sera converti en
// fr/unice/C.class
String nom = convertirNom(nomClasse);
byte octets[] = lireOctetsClasse(nom);
Class c = defineClass(nomClasse, octets,
                      0, octets.length);
return c;
```


Méthode **`defineClass`**

- C'est la méthode « magique » qui transforme un tableau d'octets en classe Java
- Elle a aussi la tâche de ranger la nouvelle classe Java dans un cache pour pouvoir la retrouver si on demande de la recharger ensuite
- Cette méthode est une méthode **`native`** de la classe **`ClassLoader`**

- En fait le code précédent de la méthode **`findClass`** est un peu simplifié
- On ne pourra utiliser la police de sécurité avec les classes chargées par un tel chargeur car le codesource de la classe ne sera pas connue de la JVM
- Il faudrait utiliser une variante de la méthode **`define`** qui prend en paramètre le codesource de la classe

Méthode defineClass

- La méthode **`defineClass`** peut avoir un dernier paramètre de type **`CodeSource`** qui permet d'indiquer l'origine de la classe (et les certificats pour la vérification de la signature, si la classe est signée)
- Ce **`CodeSource`** joue un rôle très important pour la sécurité (voir le cours sur la sécurité)
- Si on ne précise pas ce paramètre, la classe est associée à un *codeSource* d'URL et à des certificats **`null`**

L'exemple de URLClassLoader

- Un bon conseil pour ceux qui veulent écrire leur propre chargeur de classes : lire le code source de la méthode **`findClass`** de la classe **`URLClassLoader`** (tout n'est malheureusement pas si simple !)

Méthodes findResource et findResources

- On peut redéfinir ces 2 méthodes protected pour indiquer comment le chargeur de classe recherche les ressources
- Elles renvoient un URL ou une Enumeration (d'URL)

Sécurité et chargeurs de classes

Permission pour créer un chargeur de classes

- Il est évidemment très dangereux de donner la possibilité à une classe de créer un nouveau chargeur de classes
- La classe doit avoir la permission **`java.lang.RuntimePermission`** avec le but **`createClassLoader`**

Pourquoi c'est dangereux

- La méthode **`protected defineClass`** de la classe **`ClassLoader`** associe un domaine de protection aux classes chargées
- Dans ce domaine de protection il peut associer toutes les permissions qu'il veut, y compris, par exemple, la permission d'écrire n'importe où dans le disque dur local

Compléments

Les pièges de la délégation au parent

- Le comportement standard préconisé pour l'écriture des chargeurs de classes (délégation au parent) ne convient pas toujours et peut comporter des pièges
- Ainsi, soit une classe extension (placée dans le répertoire **`lib/ext`**) charge une classe **`C`** par la méthode **`forName(String nom)`**
- Bien que la classe **`C`** soit placée dans le *classpath*, elle n'est pas trouvée lors de l'exécution
- En effet, la classe est chargée par le chargeur des extensions qui ne recherche pas dans le *classpath* (ni son parent qui est le chargeur interne)

Une solution (1)

- Une solution pour ce problème : utiliser la méthode **`Thread.getContextClassLoader`** qui renvoie le chargeur utilisé pour charger le créateur du thread en cours
- Le plus souvent, c'est le chargeur utilisé pour la classe principale de l'application, c'est-à-dire le chargeur système qui recherche dans le *classpath*
- En effet, pour qu'un nouveau *thread* le change, il faut qu'il utilise la méthode **`setContextClassLoader`**, ce qui est rare

Une solution (2)

- On utilise alors la méthode **`Class.forName`** à 3 paramètres ou la méthode **`ClassLoader.loadClass`** pour faire charger la classe **`C`** par le chargeur renvoyé par **`getSystemClassLoader`**

Déboguer

- L'option **`-verbose:class`** de la commande java fait afficher des informations au moment où les classes sont chargées

Recharger une classe

- Il peut être intéressant de recharger une nouvelle version à chaud d'une classe
- Mais dans la version actuelle de la JVM, il n'est pas possible de recharger une classe déjà chargée dans la JVM
- En effet, il est impossible
  - d'avoir 2 versions d'une même classe dans la JVM
  - de décharger de la JVM une classe déjà chargée

Recharger une classe

- Mais on sait que 2 classes chargées par 2 chargeurs de classes différents sont considérées comme différentes par la JVM
- Il suffit donc de charger la nouvelle version avec un nouveau chargeur de classes
- On crée alors des instances de cette nouvelle classe pour effectuer les nouvelles fonctionnalités de la nouvelle version
- En ce cas, il faut être conscient que les références obtenues auparavant à des instances de cette classe utiliseront l'ancienne version de la classe

`getSystemClassLoader`

- Méthode de **`ClassLoader`** qui renvoie le chargeur système qui charge les classes depuis le *classpath*
- Par exemple, si le *classpath* contient le répertoire **`rep`**, et si on recherche la classe **`fr.unice.Classe`**, le fichier **`Classe.class`** sera recherchée dans le répertoire **`rep/fr/unice`**

`getContextClassLoader`

- Méthode de la classe **`Thread`** qui récupère le chargeur associé au *thread* en cours
- C'est le plus souvent le chargeur système

Différence entre **`Class.forName`** et **`ClassLoader.loadClass`**

- Il existe une petite différence entre ces 2 appels : **`Class.forName`** peut renvoyer des « classes système » associées à des types spéciaux comme « **`[String`** » qui correspond au type « tableau de **`String`** »

Bibliographie

- Article ********