Programmation avancée et répartie en Java : interfaçage avec d'autres langages

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de M1 MIAGE
(d'après les notes de cours de Fabrice Mourlin)

Plan

Plan

Déroulement du cours

Qu'est-ce ?

Ce cours introduit l'API Java Native Interface (JNI) qui permet d'étendre JAVA avec du code compilé écrit en C ou C++ :

Nous verrons :

- Pourquoi réaliser du code natif ?
- Les phases de génération : un exemple simple
- Les caractéristiques générales de JNI
- Exemples : emploi des types Java en C, accès aux attributs des objets JAVA, création d'instances.
- Le problème de l'intégrité des références aux objets JAVA.
- Le traitement des exceptions JAVA

Nous supposons :

Que vous sachiez écrire/lire des lignes simples de C. De même, pour la prochaine section, connaître SQL.

Pourquoi réaliser du code natif ?

Il y a des situations dans laquelle le code ne peut pas être complétement écrit en JAVA :

- Une grande quantité de code compilé existe déja et fonctionne de manière satisfaisante. La fourniture d'une interface avec JAVA peut être plus intéressante qu'une réécriture complète.
- Une application doit utiliser des services non fournis par JAVA (et en particulier pour exploiter des spécificités de la plateforme d'exécution. Exemple : accès à des cartes).
- Le code JAVA n'est pas adapté à une partie critique de l'application et la réalisation dans un code natif serait plus efficiente.

Il est possible d'implanter en JAVA des méthodes natives réalisées typiquement en C ou C++. Il existe diverses manières d'assurer cette liaison code compilé-Java. Depuis la version JAVA 1.1 le protocole JNI a été défini pour rendre cette adaptation plus facilement indépendante de la réalisation de la machine virtuelle.

Inconvénients

Une classe comprenant des méthodes natives ne peut pas être téléchargée au travers du réseau de manière standard : il faudrait que le serveur ait connaissance des spécificités de la plate-forme du client. De plus une telle classe ne peut faire appel aux services de sécurité de JAVA.

Bien entendu pour toute application JAVA qui s'appuie sur des composants natifs on doit réaliser un portage du code natif sur chaque plate-forme spécifique. De plus c'est un code potentiellement plus fragile puisque les contrôles (pointeurs, taille, etc.) et la récupération d'erreurs sont entièrement sous la responsabilité du programmeur.

Aussi, il est également possible d'exécuter du code JAVA au sein d'une application écrite en C/C++ en appelant directement la machine virtuelle (JAVA enchassé dans C).

L'exemple "hello world" (1)

Les phases

1. Création d'une classe "HelloWorld" avec une méthode native.
2. Compilation du code par javac
3. Génération du fichier d'inclusion C (utilitaire javah)
4. Écriture du code natif en C (".c")
5. Création d'une librairie dynamique
6. Exécution avec "java"

javah fournit une définition d'un en-tête de fonction C pour la réalisation de la méthode native getGreetings() définie dans la classe HelloWorld. Écriture d'un fichier C qui réalise le code et fait appel à des fonctions et des types prédéfinis de JNI. Utilisation du compilateur C pour générer une librairie dynamique à partir des fichiers .c et .h définis ci-dessus. (sous Windows une librairie dynamique est une DLL).

(1) Écriture d'une classe avec code natif

```
package hi; // Pas obligatoire
class HelloWorld {
 static {System.loadLibrary("hello");}
 public static native String getGreetings();
 public static void main (String[] tArgs) {
     for (int ix = 0 ; ix < tArgs.length; ix++) {
        System.out.println(getGreetings() + tArgs[ix]);
 ...
```

Une méthode marquée **native** ne dispose pas de corps. Comme pour une méthode **abstract** le reste de la "signature" de la méthode (arguments, résultat,...) doit être spécifié.

Le bloc **static** est exécuté au moment du chargement de la classe. A ce moment il provoque alors le chargement d'une bibliothèque dynamique contenant le code exécutable natif lié à la classe. Le système utilise un moyen standard (mais spécifique à la plate-forme) pour faire correspondre le nom hello à un nom de bibliothèque (libhello.so sur Linux, hello.dll sur Windows).

(2) Création des binaires JAVA de référence

La classe ainsi définie se compile comme une autre classe :

```
javac -d . HelloWorld.java
```

Le binaire Java généré est exploité par les autres utilitaires employés dans la suite de ce processus. Dans l'exemple on aura un fichier "HelloWorld.class" situé dans le sousrépertoire "hi" du répertoire ciblé par l'option "-d" (ici le dossier courant).

(3) Génération du fichier d'inclusion C

"javah" va permettre de générer à partir du binaire Java un fichier d'inclusion C/C++ "hi_HelloWorld.h" qui définit les prototypes des fonctions afin de réaliser la méthode native de "HelloWorld"

```
javah -d . -jni hi.HelloWorld
```

```
#include <jni.h>
/* Header for class hi_HelloWorld */
#ifndef _Included_hi_HelloWorld
#define _Included_hi_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
...
JNIEXPORT jstring JNICALL Java_hi_HelloWorld_getGreetings (JNIEnv *, jclass);
#ifdef __cplusplus
}
```

On notera que la fonction rend l'équivallent d'un type Java (jstring) et, bien qu'étant définie sans paramètres en JAVA, comporte deux paramètres en C : le pointeur d'interface "JNIEnv" permet d'accéder aux objets "Java" et "jclass" référence la classe courante. Dans une méthode d'instance le paramètre de type "jobject" référencerait l'instance courante.

(4) Écriture du code natif

On définit un fichier source C “hi_HelloWorldImp.c” :

```
#include <jni.h>
#include "hi_HelloWorld.h"
JNIEXPORT jstring JNICALL Java_hi_HelloWorld_getGreetings(JNIEnv *env , jclass curclass) {
  ...
  return (*env)->NewStringUTF(env, "Hello ");
}
```

env nous fournit une fonction NewStringUTF qui nous permet de générer une chaîne Java à partir d’une chaîne C.

(5) Création d'une librairie dynamique

Exemple sous Linux (le ".h" dans le répertoire courant) :

```
#!/bin/sh
# changer DIR en fonction des besoins
DIR=/usr/local/java
cc -G -I$DIR/include
      -I$DIR/include/linux
   hi_HelloWorldImp.c -o libhello.so
```

Exemple de génération sous Windows avec VisualC++ :

```
cl -Ic:\java\include
   -Ic:\java\include\win32
 -LD hi_HelloWorldImp.c -Fehello.dll
```

(6) Éxecution

```
java hi.HelloWorld World underWorld
  Hello World
  Hello underWorld
```

Si, par contre, vous obtenez une exception ou un message indiquant que le système n'a pas pu charger la librairie dynamique il faut positionner correctement les chemins d'accès aux librairies dynamiques ("LD_LIBRARY_PATH" sous Linux).

JNI et GC

Les fonctions de JNI sont adressables au travers d'un environnement (pointeur d'interface vers un tableau de fonctions) spécifique à un thread. C'est la VM elle-même qui passe la réalisation concrète de ce tableau de fonctions et assure ainsi la compatibilité binaire des codes natifs. En particulier :

- Créer, consulter, mettre à jour des objets Java, (et opérer sur leurs verrous). Opérer avec des types natifs Java.
- Appeler des méthodes Java
- Manipuler des exceptions
- Charger des classes et inspecter leur contenu

Le point le plus délicat dans ce partage de données entre C et Java et celui garbage collector : il faut se protéger contre des déréférencements d'objets ou contre des effets de compactage en mémoire (déplacements d'adresses), mais il faut savoir aussi faciliter le travail du glaneur pour recupérer de la mémoire.

Types, accès aux membres, création d'objets (1)

```
package hi;
class Uni {
 static {System.loadLibrary("uni");}
 public String [] tb ;
 public Uni(String[] arg) {tb = arg;}
 public native String [] getMess(int n, String mess);
 public static native Uni dup(Uni other);
 public String toString() {
    String res = super.toString() ;
    // pas efficient
    for (int ix = 0 ; ix < tb.length; ix ++) {res=res+'\n'+tb[ix];}
    return res ;
 }
 public static void main (String[] tArgs) {
    Uni you = new Uni(tArgs);
    System.out.println(you);
    String[] mess = you.getMess(tArgs.length,"Hello");
    for (int ix=0; ix<mess.length; ix++) {System.out.println(mess[ix]);}
    Uni me = Uni.dup(you);
    System.out.println(me);
 }
}
```

Types, accès aux membres, création d'objets (2)

Exemple d'utilisation :

```
gava@gava-laptop:/.$java hi.Uni World
  hi.Uni@1dce0764
  World
    Hello
  hi.Uni@1dce077f
  World
```

La méthode native getMess(**int** nb, String mess) génère un tableau de chaînes contenant "nb" fois le même message mess :

```
/* Class: hi_Uni
* Method: getMess
* Signature: (ILjava/lang/String;)[Ljava/lang/String;
*/
JNIEXPORT jobjectArray JNICALL Java_hi_Uni_getMess
              (JNIEnv *env , jobject curInstance,jint nb , jstring chaine) {
  /* quelle est la classe de String ? */
  jclass stringClass = (*env)->FindClass(env,"java/lang/String");
  /* un tableau de "nb" objet de type "stringClass"
  * chaque element est initialise a "chaine" */
  return (*env)->NewObjectArray(env,(jsize)nb,stringClass,chaine);
}
```

Types, accès aux membres, création d'objets (3)

- La fonction NewObjectArray permet la création d'objets Java. Elle doit connaître le type de ses composants (ici fourni par stringClass). L'initialisation de chaque membre d'un tel tableau se fait par l'accesseur SetObjectArrayElement() (mais ici on profite du paramètre d'initialisation par défaut)
- JNI fournit des types C prédéfinis pour représenter des types primitifs Java (jint) ou pour des objets (jobject, jstring,...)
- La fonction FindClass permet d'initialiser le bon paramètre désignant la classe java.lang.String (la notation utilise le séparateur "/" !). Noter également la représentation de la signature de la fonction getMess : (ILjava/lang/String ;) indique un premier paramètre de type int (symbolisé par la lettre I) suivi d'un objet (lettre L+type+ ;). De même [Ljava/lang/String ; désigne un résultat qui est un tableau à une dimension (lettre [) contenant des chaînes.

Types, accès aux membres, création d'objets (4)

La méthode statique "dup" clone l'instance passée en paramètre :

```
/* Class: hi_Uni; Method: dup
* Signature: (Lhi/Uni;)Lhi/Uni; */
JNIEXPORT jobject JNICALL Java_hi_Uni_dup (JNIEnv * env,jclass curClass , jobject other) {
 jfieldID idTb; jobjectArray tb;
 jmethodID idConstr;
 /* en fait inutile puisque c'est curClass !*/
 jclass uniClass = (*env)->GetObjectClass(env, other);
 if (! (idTb = (*env)->GetFieldID (env,uniClass,"tb","[Ljava/lang/String;"))) return NULL;
 tb=(jobjectArray) (*env)->GetObjectField(env,other,idTb);
 /* on initialise un nouvel objet */
 if(!(idConstr=(*env)->GetMethodID(env, curClass,"<init>", "([Ljava/lang/String;)V"))) return NULL ;
 return (*env)->NewObject(env,curClass,idConstr,tb);
}
```

La récupération du champ tb (de type tableau de chaîne) sur l'instance passée en paramètre se fait par la fonction GetObjectField. On a besoin de la classe de l'instance consultée et de l'identificateur du champ qui est calculé par GetFieldID.

Types, accès aux membres, création d'objets (5)

De la même manière l'appel d'une méthode nécessite une classe et un identificateur de méthode calculé par GetMethodID. Ici ce n'est pas une méthode qui est appelée mais un constructeur et l'identifiant est calculé de manière particulière (<init>), le type indique un paramètre de type tableau de chaîne et un "résultat" qui est **void** (lettre V).

JNI fournit ainsi des accesseurs à des champs (Get<**static**><type>Field, Set<**static**><type>Field) et des moyens d'appeler des méthodes (Call<statut><type>Method ; exemple CallStaticBooleanMethod). Il existe, en plus, des méthodes spécifiques aux tableaux et aux Strings.

Références sur des objets Java (1)

Le passage du glaneur de mémoire sur des objets Java pourrait avoir pour effet de rendre leur référence invalide ou de les déplacer en mémoire. Les références d'objet Java transmises dans les transitions vers le code natif sont protégées contre les invalidations (elles redeviennent récupérables à la sortie du code natif).

Toutefois JNI autorise le programmeur à explicitement rendre une référence locale récupérable. Inversement il peut aussi se livrer à des opérations de "punaisage" (pinning) lorsque, pour des raisons de performances, il veut pouvoir accéder directement à une zone mémoire protégée :

```
const char * str = (*env)->GetStringUTFChars(env,javaString,0);
.... /* opérations sur "str" */
(*env)->ReleaseStringUTFChars(env, javaString, str);
```

Références sur des objets Java (1)

Des techniques analogues existent pour les tableaux de scalaires primitifs (int, float, etc.). Il est essentiel que le programmeur C libère ensuite la mémoire ainsi gelée. Si on veut éviter de bloquer entièrement un tableau alors qu'on veut opérer sur une portion de ce tableau , on peut utiliser des fonctions comme :

```
void GetIntArrayRegion(JNIenv* env, jintArray tableau,jsize debut, jsize taille, jint * buffer);
void SetIntArrayRegion(....
```

Le programmeur a aussi la possibilité de créer des références globales sur des objets Java. De telles références ne sont pas récupérables par le glaneur à la sortie du code natif, elles peuvent être utilisées par plusieurs fonctions implantant des méthodes natives. Ici aussi il est de la responsabilité du programmeur de libérer ces références globales.

Les exceptions et JNI

JNI permet de déclencher une exception quelconque ou de recupérer une exception Java provoquée par un appel à une fonction JNI. Une exception Java non récupérée par le code natif sera retransmise à la machine virtuelle.

```
....
jthrowable exc;
(*env)->CallVoidMethod(env, instance , methodID);
/* quelque chose s'est-il produit? */
exc = (*env)->ExceptionOccurred(env);
if (exc) {
  jclass NouvelleException ;
  ... /* diagnostic */
  /* on fait le ménage */
  (*env)->ExceptionClear(env) ;
  /* et on fait du neuf ! */
  NouvelleException = (*env)->FindClass(env,"java/lang/IllegalArgumentException");
  (*env)->ThrowNew(env,NouvelleException, message);
}
...
```

Déroulement du cours

1. Utilisation de codes natifs : J.N.I.
2. Java et le web : les Applets Java

Qu'est-ce ? (1)

En général

Les Applets constituents des petites applications Java hébergées au sein d'une page HTML, leur code est téléchargé par le navigateur :

- Une Applet est, à la base, un panneau graphique. Un protocole particulier la lie au navigateur qui la met en oeuvre (cycle de vie de l'Applet).
- Les codes qui s'exécutent au sein d'une Applet sont soumis à des restrictions de sécurité.

Attention

Plus de détails dans le prochain cours : archi WEB !

Qu'est-ce ? (2)

Une Appliquette (Applet) est une portion de code Java qui s'exécute dans l'environnement d'un navigateur au sein d'une page HTML. Elle diffère d'une application par son mode de lancement et son contexte d'exécution.

Une application autonome est associée au lancement d'un processus JVM qui invoque la méthode main d'une classe de démarrage.

Une JVM associée à un navigateur peut gérer plusieurs Applets, gérer leur contexte (éventuellement différents) et gérer leur "cycle de vie" (initialisation, phases d'activité, fin de vie).

Une applet s'exécutant dans le cadre d'une "page" HTML, la requête de lancement et les paramètres associés sont en fait contenus dans le fichier source décrivant la page.

Qu'est-ce ? (3)

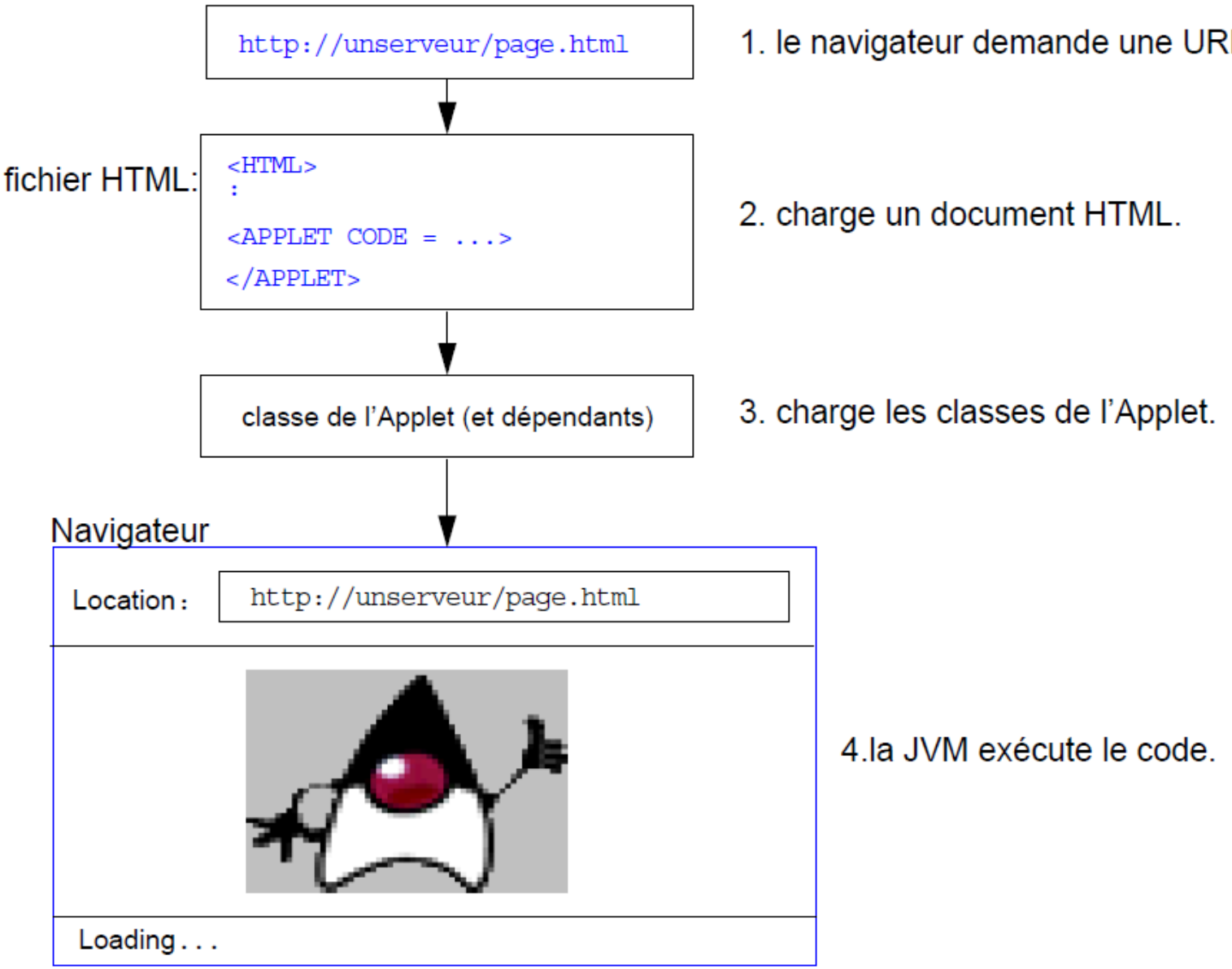
Le lancement d'une Applet suppose donc :

- La désignation d'un document au navigateur au travers d'une adresse URL
- Le chargement et la mise en page du document HTML associé
- Le chargement du code de l'Applet spécifié dans le document (et éventuellement le chargement des codes des classes distantes appelées par l'Applet).
- La gestion, par la JVM du navigateur, du cycle de vie de l'Applet au sein du document mis en page.

Exemple d'utilisation de la balise APPLET (`OBJECT` en HTML 4) :

```
<P> et maintenant notre Applet :
 <APPLET code=‘‘fr.gibis.applets.MonApplet.class’’
         width=100 height=100></APPLET>
```

Lancement


http://unserveur/page.html
1. le navigateur demande une URL.
fichier HTML:
<HTML>
:
<APPLET CODE = ...>
</APPLET>
2. charge un document HTML.
classe de l'Applet (et dépendants)
3. charge les classes de l'Applet.
Navigateur
Location :
http://unserveur/page.html
4.la JVM exécute le code.
Loading . . .

Restriction de sécurité (1)

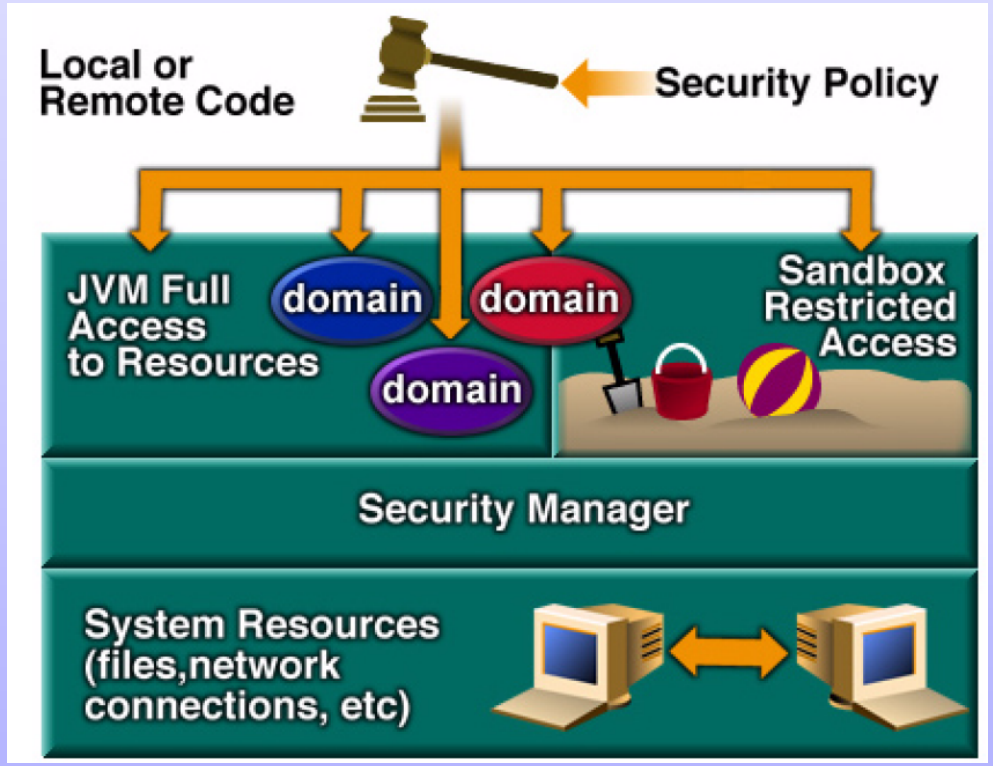
Dans ce contexte on va exécuter des codes de classe : classses “distantes” et classes “système” de la librairie java locale. Pour éviter qu'un code distant puisse réaliser des opérations contraires à une politique de sécurité élémentaire, des règles par défaut s'appliquent (sandbox security policy) :

- L'Applet ne peut obtenir des informations sur le système courant (hormis quelques informations élémentaires comme la nature du système d'exploitation, le type de JVM, etc.).
- L'Applet ne peut connaître le système de fichiers local (et donc ne peut réaliser d'entrée/sortie sur des fichiers).
- L'Applet ne peut pas lancer de processus
- Les communications sur réseau (par socket) ne peuvent être établies qu'avec le site d'origine du code de l'Applet.

Bien entendu un code d'Applet ne peut pas contenir de code “natif” (par essence non portable).

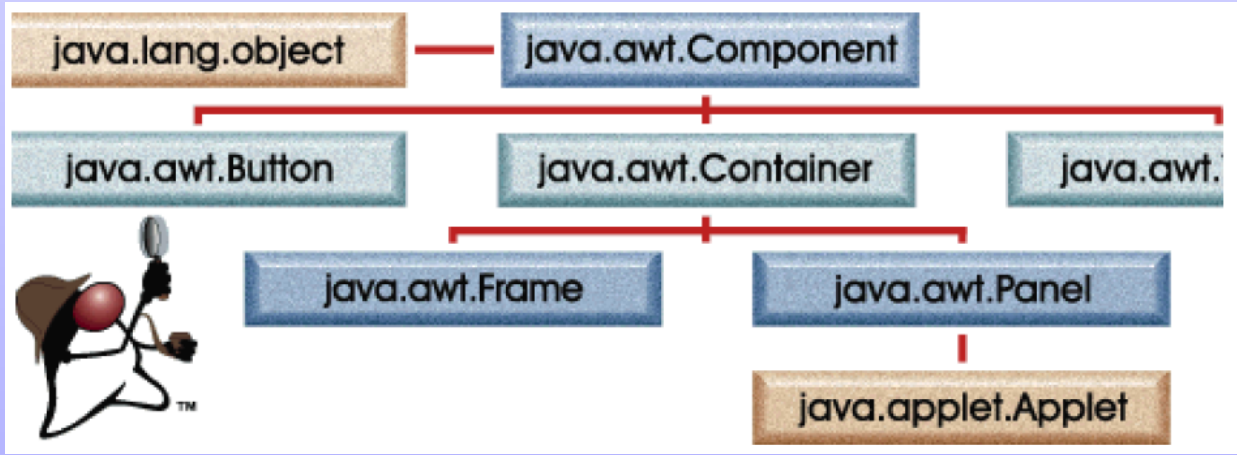
Restriction de sécurité (2)

Un système standard de mise en place de domaines de sécurité (Protection Domain) permet d'assouplir ces règles pour permettre certaines opérations à des codes dûment authentifiés venus de sites connus. On peut ainsi imaginer que l'Applet qui vous permet de consulter votre compte en banque vous permet aussi de rapatrier des données à un endroit du système de fichier défini par vous.



Hiérarchie

L'incorporation d'une Applet à une présentation graphique (la "page" du navigateur) fait qu'une Applet est, de manière inhérente, une classe graphique (même si on peut créer des Applets non-graphiques). Cela revient à signaler au navigateur qu'une zone de dimensions données n'est plus gérée directement par l'interpréteur de HTML mais par un programme autonome qui prend le contrôle de la représentation graphique de cette zone. Toute classe qui est une Applet doit dériver de java.applet.Applet qui est elle-même une classe dérivée de java.awt.Panel qui représente un "panneau" graphique dans la librairie graphique AWT.



Méthodes (1)

Une Applet étant un Container graphique, elle hérite des méthodes qui permettent de disposer des composants graphiques dans la zone gérée par la classe. Ce point est traité du point de vue de la programmation AWT “classique”, et, d’un autre coté , nous allons nous attacher à trois groupes de méthodes :

1. méthodes appelées par le système graphique pour la notification d’une demande de rafraîchissement d’une zone de l’écran : repaint(), update(Graphics), paint(Graphics). Dans le cas où l’Applet souhaite gérer le graphique de bas niveau (au lieu de laisser agir le système automatique attachés à des composant AWT de haut niveau) elle doit conserver un modèle logique des éléments graphiques qu’elle gère et être capable de les redessiner à la demande.

Méthodes (2)

1. Méthodes spécifiques aux Applets et qui leur permettent de demander au navigateur des informations ou des actions liées au contexte : informations sur la page HTML courante, chargement d'une ressource sur le site d'origine de la page, etc.
2. Méthodes spécifiques aux Applets et à leur cycle de vie : le navigateur doit pouvoir notifier à l'Applet qu'il veut l'initialiser (init()), qu'il veut la rendre active ou inactive (start(), stop()), ou qu'il veut la "détruire" (destroy()). Par défaut ces méthodes ne font rien et il faut en redéfinir certaines d'entre elles pour obtenir un comportement significatif de l'Applet.

Les balises (1)

```
<APPLET
 [archive=ListeArchives]
 code=package.NomApplet.class
 width=pixels height=pixels
 [codebase=codebaseURL] [alt=TexteAlternatif]
 [name=nomInstance] [align=alignement]
 [vspace=pixels] [hspace=pixels]
>
   [<PARAM name=Attribut1 value=valeur>]
   [<PARAM name=Attribut2 value=valeur>]
   ...
   [HTMLalternatif]
</APPLET>
```

en HTML 4 :

```
<OBJECT codetype="application/java" classid="MonApplet.class"
        width=300 height=400>
</OBJECT>
```

Les balises (2)

La balise HTML APPLET comporte les attributs suivants :

- code=appletFile.class — Cet attribut obligatoire fournit le nom du fichier contenant la classe compilée de l'applet (dérivée de java.applet.Applet). Son format pourrait également être a Package.appletFile.**class**.. La localisation de ce fichier est relative à l'URL de base du fichier HTML de chargement de l'applet.
- width=pixels height=pixels — Ces attributs obligatoires fournissent la largeur et la hauteur initiales (en pixels) de la zone d'affichage de l'applet, sans compter les éventuelles fenêtres ou boîtes de dialogue affichées par l'Applet.
- codebase=codebaseURL — Cet attribut facultatif indique l'URL de base de l'applet : le répertoire contenant le code de l'applet. Si cet attribut n'est pas précisé, c'est l'URL du document qui est utilisé.

Les balises (3)

La balise HTML APPLET comporte les attributs suivants :

- name=appletInstanceName — Cet attribut, facultatif, fournit un nom pour l'instance de l'applet et permet de ce fait aux applets situées sur la même page de se rechercher mutuellement (et de communiquer entre-elles).
- archive=ListeArchives permet de spécifier une liste de fichiers archive .jar contenant les classes exécutables et, éventuellement des ressources. Les noms des archives sont séparés par des virgules. Ce point ne sera pas abordé dans ce cours.
- object=objectFile.ser permet de spécifier une instance d'objet à charger. Ce point ne sera pas abordé dans ce cours.
- `<param name=appletAttribute1 value=value>` — Ces éléments permettent de spécifier un paramètre à l'applet. Les applets accèdent à leurs paramètres par la méthode getParameter().

Système graphique de bas niveau

Applet héritant de Panel il est possible de modifier l'aspect en utilisant les méthodes graphiques des composants. Voici un exemple minimal d'Applet qui écrit du texte de manière graphique :

```
import java.awt.* ;
import java.applet.Applet ;
public class HelloWorld extends Applet {
 public void paint(Graphics gr) {gr.drawString("Hello World?", 25 ,25) ;}
}
```

Les arguments numériques de la méthode drawString() sont les coordonnées x et y du début du texte. Ces coordonnées font référence à la "ligne de base" de la police. Mettre la coordonnée y à zero aurait fait disparaître la majeure partie du texte en haut de l'affichage (à l'exception des parties descendantes des lettres comme "p", "q" etc.)

Méthodes d'accès aux ressources de l'environnement

Récupération de paramètres. Exemple de source HTML :

```
<APPLET code="Dessin.class" width=200 height=200>
   <PARAM name="image" value="duke.gif"> </APPLET>
```

Le code Java correspondant :

```
public class Dessin extends Applet {
 Image img ;
 public void init() {// redef. méthode standard cycle vie
   String nomImage = getParameter("image") ;
   if ( null != nomImage) {img = getImage(getDocumentBase(),nomImage);}
 }
 public void paint (Graphics gr) {
  if( img != null) {gr.drawImage(img,50,50, this);}
   else {gr.drawString("image non chargée",25,25);}
```

Les méthodes de chargement de media comme getImage() sont des méthodes asynchrones. On revient de l'appel de la méthode alors que le chargement est en cours (et, possiblement, non terminé). Il est possible que paint() soit appelé plusieurs fois au fur et à mesure que l'image devient complètement disponible.

Cycle de vie (1)

La méthode init() est appelée pour que l'applet soit créée et chargée pour la première fois dans un navigateur (firefox ou AppletViewer). L'applet peut utiliser cette méthode pour initialiser les valeurs des données. Cette méthode n'est pas appelée chaque fois que le navigateur ouvre la page contenant l'applet, mais seulement la première fois juste pour le changement.

La méthode start() est appelée pour indiquer que l'applet doit être "activée". Cette situation se produit au démarrage de l'applet, une fois la méthode init() terminée. Elle se produit également lorsque la fenêtre courante du navigateur est restaurée après avoir été iconisée ou lorsque la page qui héberge l'applet redevient la page courante du navigateur. Cela signifie que l'applet peut utiliser cette méthode pour effectuer des tâches comme démarrer une animation ou jouer des sons.

```
public void start() {musicClip.play();}
```

Cycle de vie (2)

La méthode stop() est appelée lorsque l'applet cesse d'être en phase active. Cette situation se produit lorsque le navigateur est icônisé ou lorsque le navigateur présente une autre page que la page courante. L'applet peut utiliser cette méthode pour effectuer des tâches telles que l'arrêt d'une animation.

```
public void stop() {musicClip.stop();}
```

Les méthodes start() et stop() forment en fait une paire, de sorte que start() peut servir à déclencher un comportement dans l'applet et stop() à désactiver ce comportement.

La méthode destroy() est appelée avant que l'objet applet ne soit détruit c.-à-d. enlevé du cache du navigateur.

Cycle de vie (3)

```
// Suppose l'existence du fichier son "cuckoo.wav"
// dans le répertoire "sounds" situé dans
// le répertoire du fichier HTML d'origine
import java.awt.Graphics;
import java.applet.*;
public class HwLoop extends Applet {
 AudioClip sound;
 // attention syntaxe d'URL!!!
 public void init() {sound = getAudioClip(getDocumentBase(),"sounds/cuckoo.wav");}
 // méthode de dessin de java.awt.Graphics
 public void paint(Graphics gr) {gr.drawString("Audio Test", 25, 25);}
 public void start () {sound.loop();}
 public void stop() {sound.stop();}
}
```

Au travail !