

Le chargement des classes

contenu de la section

LE CHARGEMENT DES CLASSES.....1

CONTENU DE LA SECTION.....2

QUELQUES GÉNÉRALITÉS SUR LE CHARGEMENT DES CLASSES.....4

les composants de l'architecture des JVM4

LES CHARGEURS D'UNE JVM (1).....5

le modèle général de chargement.....5

l'organisation des chargeurs dans les plates-formes5

LES CHARGEURS D'UNE JVM (2).....6

un exemple.....6

LA CONCEPTION DES CHARGEURS (1).....7

la classe java.lang.ClassLoader.....7

LA CONCEPTION DES CHARGEURS (2).....8

la structure générale de la méthode loadClass().....8

LA CONCEPTION DES CHARGEURS (3).....9

structure générale de la méthode loadClass() suite.....9

LA CONCEPTION DES CHARGEURS (4).....10

un exemple simple de chargeur pour classes "cryptées".....10

QUELQUES ERREURS (1).....11

la non délégation11

QUELQUES ERREURS (2).....12

la duplication des classes12

QUELQUES ERREURS (3).....13

Une violation de contrainte liée à la délégation13

QUELQUES ERREURS (4).....14

l'interblocage14

LA CONCEPTION DES CHARGEURS (5).....15

la manipulation du byte code.....15

les librairies et outils associés.....15

UNE INTRODUCTION À JAVASSIST.....16

la lecture-écriture du byte code.....16

la modification du byte code.....16

LA CONCEPTION DES CHARGEURS (6).....17

Un exemple simple de chargeur pour instrumenter les méthodes annotées par @instrument.....17

LES CHARGEURS ET LE TYPAGE (1).....18

La relation de sous-typage.....18

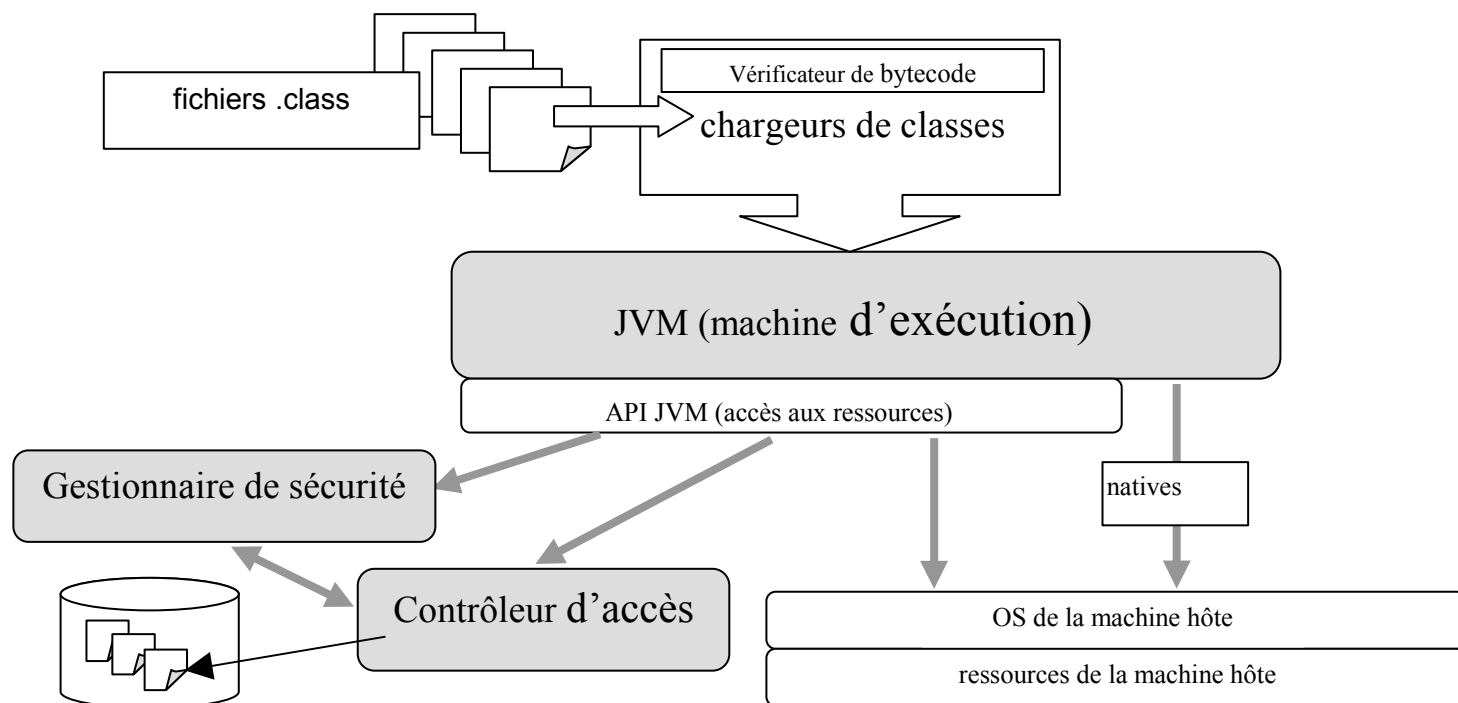
LES CHARGEURS ET LE TYPAGE (2).....19

un exemple (suite).....19

java avancé	chapitre 12
L'UTILISATION D'UN CHARGEUR SPÉCIFIQUE.....	20
<i>une contrainte</i>	20
<i>un exemple simple</i>	20
LE DÉCHARGEMENT DES CLASSES.....	21
<i>la spécification de java</i>	21
QUELQUES GÉNÉRALITÉS SUR L'OPTIMISATION DU CHARGEMENT DES CLASSES	22
<i>quelques règles de bon sens</i>	22
<i>quelques solutions génériques</i>	22
ÉVITER LE CHARGEMENT DE CLASSES INUTILES.....	23
<i>le principe</i>	23
ECONOMISER LA MÉMOIRE (1).....	24
<i>le principe</i>	24
ECONOMISER LA MÉMOIRE (2).....	25
<i>l'utilisation de code "trampoline"</i>	25
<i>la classe Proxy et l'interface InvocationHandler</i>	25
ECONOMISER LA MÉMOIRE ET RÉDUIRE LA DURÉE DE CHARGEMENT.....	26
<i>l'utilisation d'obfuscateurs</i>	26

Quelques généralités sur le chargement des classes

les composants de l'architecture des JVM



- chargement agressif vs chargement paresseux
- visualisation du chargement via l'option **-verbose:class**
- existence de plusieurs chargeurs de classes
- possibilité de rajouter des chargeurs spécifiques

Les chargeurs d'une JVM (1)

le modèle général de chargement

- les chargeurs de classe sont organisés selon un arbre
- tout chargeur commence par déléguer le chargement à son chargeur parent. Il n'intervient qu'en cas d'échec du parent

l'organisation des chargeurs dans les plates-formes

- **depuis JDK1.2**
 - un chargeur primordial écrit en C intégré à la JVM (unique et non modifiable)
 - racine de l'arbre de chargement
 - charge les classes de boot (rt.jar)
 - un chargeur prédéfini **sun.misc.Launcher\$ExtClassLoader** pour :
 - descendant du chargeur primordial
 - les classes des .jar des répertoires d'extension std
 - un chargeur prédéfini **sun.misc.Launcher\$AppClassLoader** pour :
 - descendant de **sun.misc.Launcher\$ExtClassLoader**
 - les classes accessibles via le CLASSPATH
 - des chargeurs spécifiques (*RMIClassLoader*, *AppletClassLoader*, ... installés par les applications durant l'exécution)

sa référence est null
pas de vérification de byte code

sa référence est null
pas de vérification de byte code

sa référence est **ClassLoader.getSystemClassLoader()**
vérification de byte code

en JDK 1.1 un chargeur intégré à la JVM (unique et non modifiable) pour
les classes de l'API java + les classes accessibles via le CLASSPATH +
des chargeurs spécifiques (installés par les applications durant l'exécution)

Les chargeurs d'une JVM (2)

un exemple

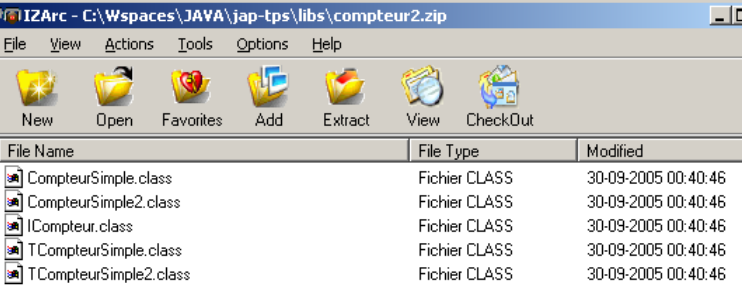
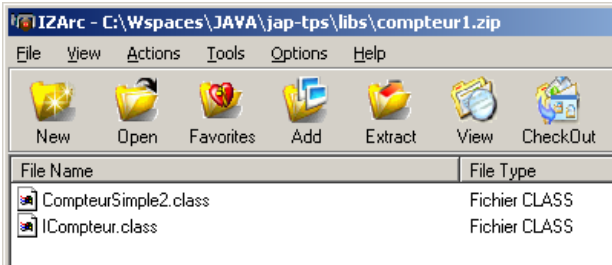
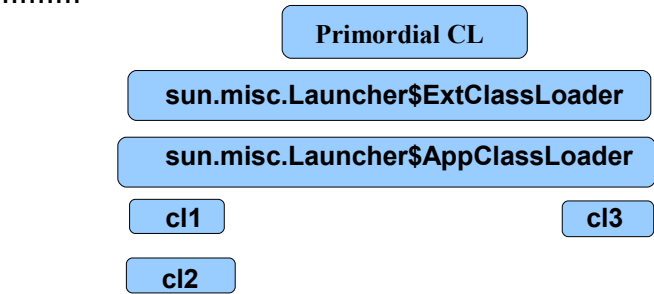
```
.....
public class TestClassLoader1 {
    .....

    public static void main(String[] args) {
    try {

        URL[] urls1 = new URL[] { new URL("file:///C:/Wspaces/JAVA/jap-tps/libs/compteur1.zip"); };
        URL[] urls2 = new URL[] { new URL("file:///C:/Wspaces/JAVA/jap-tps/libs/compteur2.zip"); };
        ClassLoader cl1 = new URLClassLoader1(urls1);
        ClassLoader cl2 = new URLClassLoader1(urls2, cl1);
        ClassLoader cl3 = new URLClassLoader1(urls2);

        Class clz1 = cl1.loadClass("tp02.ICompteur");
        Class clz2 = cl2.loadClass("tp02.ICompteur");
        Class clz2b = cl2.loadClass("tp02.CompteurSimple");
        Class clz3 = cl3.loadClass("tp02.ICompteur");
        Class clz3b = cl3.loadClass("tp02.CompteurSimple");

    }
    .....
}
```



La conception des chargeurs (1)

la classe java.lang.ClassLoader

```
public abstract class ClassLoader {

    protected ClassLoader( ) throws SecurityException;
    protected ClassLoader( ClassLoader parent) throws SecurityException;
    //Les constructeurs testent que la classe invocatrice a le droit de créer un nouveau chargeur (checkCreateClassLoader)

    protected Class loadClass(String name,boolean resolve) throws ClassNotFoundException;
    //retourne la classe demandée. Si resolve est vraie la classe doit être retournée liée

    protected final void resolveClass(Class c) throws NullPointerException;

    protected final Class findSystemClass(String name) throws ClassNotFoundException;
    //demande le chargement via les chargeurs prédéfinies

    protected final Class findLoadedClass(String name) throws ClassNotFoundException;
    //recherche un objet Class pour la classe (qui a donc déjà été chargée)

    protected final Class defineClass(String name, byte data[ ],int offset,int len) throws IndexOutOfBoundsException,ClassFormatError;
    //exécute le vérificateur de bytecode puis construit un objet Class (à partir du tableau d'octets (contenu d'un .class))

    protected final ClassLoader getParent() ;
    .....
}
```

La conception des chargeurs (2)

la structure générale de la méthode loadClass()

```
import java.lang.* ;

public class NewLoader extends ClassLoader {
    public NewLoader(ClassLoader pLoader) { super(pLoader) ; }
    public NewLoader() { super() ; }

    public Class loadClass(String name,boolean resolve) throws ClassNotFoundException {
        Class c ;

        try {
            return findLoadedClass(name);
        } catch(ClassNotFoundException ex) { }

        //retrouver la classe si elle a déjà été chargée.

        int index= name.lastIndexOf('.') ;
        String packageName=(index>0) ?name.substring(0,index) : " " ;

        SecurityManager sm=System.getSecurityManager() ;
        if (sm !=null) { if (index>0) sm.checkPackageAccess(packageName) ; }

        //tester si on a le droit d'accéder au package de la classe


        try {
            ClassLoader parent = getParent();
            if (parent != null) return parent.loadClass(name,resolve) ;
            //obtenir la classe à partir d'un chargeur parent
            else return findSystemClass(name) ;
        } catch(ClassNotFoundException e) { }

        .....
    }
}
```


La conception des chargeurs (3)

structure générale de la méthode loadClass() suite

```
if (sm !=null) { if (index>0) sm.checkPackageDefinition(packageName) ; } //tester si on a le droit de créer la classe dans le package
```

```
c = findClass(name);  byte data[ ] = lookUpData(name) ; //obtenir le bytecode à charger ...  
return defineClass(name,data,0,data.length); //verifier le bytecode et créer l'objet Class
```

```
if (resolve) resolveClass(c) ; //résoudre les références,  
return c ;  
}
```

La conception des chargeurs (4)

un exemple simple de chargeur pour classes "cryptées"

```
public class SimpleCryptedClassLoader extends ClassLoader {  
    private String baseDir;  
    private byte decalage;  
  
    public SimpleCryptedClassLoader(String bD, byte dec, ClassLoader parent) { super(parent); baseDir = bD; decalage = dec; }  
    public SimpleCryptedClassLoader(String bD, byte dec) { baseDir = bD; decalage = dec; }  
  
    @Override  
    protected Class<?> findClass(String name) {  
        String fileName=name.replace('.', File.separatorChar) + ".class";  
        byte[] buf = new byte[5000];  
  
        try {  
            //lecture du contenu du fichier .class  
            File file =new File(baseDir + File.separator + fileName);  
            InputStream is=new FileInputStream(file);  
            int len = is.read(buf);  
  
            //transformation du contenu du fichier .class  
            for (int i = 0; i < len; i++) buf[i] -= decalage;  
  
            Class cl=defineClass(name,buf,0,len);  
            return cl;  
        } catch (Exception exc) { return null; }  
    }  
}
```

les classes à instrumenter ont un CLASSPATH propre

le contenu des fichiers .class a été "chiffré" au préalable par décalage de chaque octet d'une valeur "decalage"

Quelques erreurs (1)

la non délégation ...

- le ClassLoader ne peut pas trouver les classes de base (absence de délégation auprès des ClassLoaders prédéfinis)

```
public class ClassLoader1 extends ClassLoader {  
    private String baseDir;  
  
    public ClassLoader1(String bD, ClassLoader parent) { super(parent); baseDir = bD; }  
    public ClassLoader1(String bD) { baseDir = bD; }  
  
    @Override  
    public Class<?> loadClass(String name, boolean r) { return findClass(name); }
```

ce ClassLoader ne PEUT charger AUCUNE classe : il ne peut pas trouver java.lang.Object

```
    @Override  
    public Class<?> findClass(String name) {  
        String fileName=name.replace('.', File.separatorChar) + ".class";  
        byte[] buf = new byte[5000];  
  
        try {  
            //lecture du contenu du fichier .class  
            File file =new File(baseDir + File.separator + fileName);  
            InputStream is=new FileInputStream(file);  
            int len = is.read(buf);  
  
            Class cl=defineClass(name,buf,0,len);  
            if (r) resolveClass(cl);  
            return cl;  
        } catch(Exception exc) { return null; }  
    }  
}
```

Quelques erreurs (2)

la duplication des classes ...

- le ClassLoader tente de dupliquer les classes (absence de délégation auprès du cache)

```
public class ClassLoader1 extends ClassLoader {
    private String baseDir;

    public ClassLoader1(String bD, ClassLoader parent) { super(parent); baseDir = bD; }
    public ClassLoader1(String bD) { baseDir = bD; }

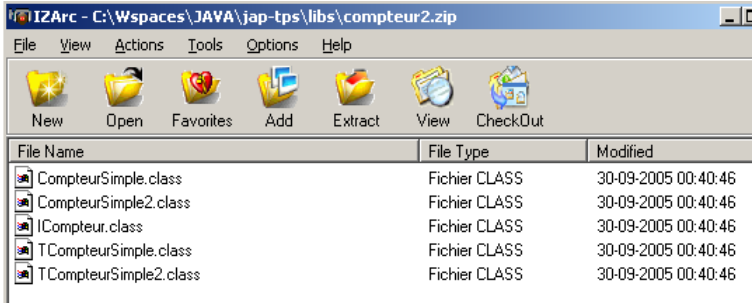
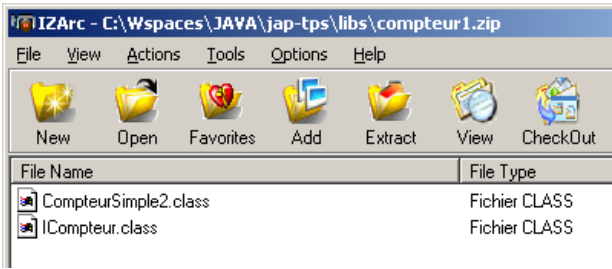
    @Override
    public Class<?> loadClass(String name, boolean r) {
        try {
            ClassLoader parent = getParent();
            if (parent != null) return parent.loadClass(name, resolve) ;           //obtenir la classe à partir d'un chargeur parent
            else return findSystemClass(name);
        } catch(ClassNotFoundException e) { }
        Class cl = findClass(name);
        if (r) resolveClass(cl);
        return cl;
    }

    @Override
    public Class<?> findClass(String name) {
        String fileName=name.replace('.', File.separatorChar) + ".class";
        byte[ ] buf = new byte[5000];

        //lecture du contenu du fichier .class
        .....
        return defineClass(name,buf,0,len);
    }
}
```

Quelques erreurs (3)

Une violation de contrainte liée à la délégation ...

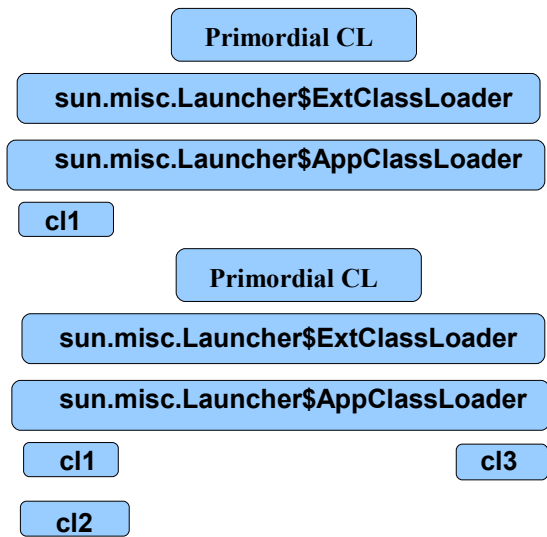


- un problème de "linkage" lié à la délégation

```
URL[] urls1 = new URL[] { new URL("file:///C:/Wspaces/jap-tps/libs/compteur1.zip"); };
ClassLoader cl1 = new URLClassLoader1(urls1);
//l'implementation CompteurSimple de l'interface ICompteur est dans le CLASSPATH
Class.forName("CompteurSimple");
```

```
URL[] urls1 = new URL[] { new URL("file:///C:/Wspaces/jap-tps/libs/compteur1.zip"); };
URL[] urls2 = new URL[] { new URL("file:///C:/Wspaces/jap-tps/libs/compteur2.zip"); };
ClassLoader cl1 = new URLClassLoader1(urls1);
ClassLoader cl2 = new URLClassLoader1(urls2, cl1);
//l'implementation CompteurSimple de l'interface ICompteur n'est pas dans le CLASSPATH
Class.forName("CompteurSimple");
```

- il est possible de définir des "buddy ClassLoader"



Quelques erreurs (4)

l'interblocage ...

- l'utilisation d'un "buddy ClassLoader" dans le cadre d'une application multithreadée peut conduire à un interblocage sur les ClassLoaders

class A extends B { }
class B extends C { }

Primordial CL

sun.misc.Launcher\$ExtClassLoader

sun.misc.Launcher\$AppClassLoader

cl1

cl2

```
//AC.zip contient les classes A et C
URL[] urls1 = new URL[] { new URL("file:///C:/Wspaces/jap-tps/libs/AC.zip"); };
//B.zip contient les classes B
URL[] urls2 = new URL[] { new URL("file:///C:/Wspaces/jap-tps/libs/B.zip"); };
ClassLoader cl1 = ClassLoader1(urls1);
ClassLoader cl2 = new URLClassLoader1(urls2, cl1);

//
new Thread(new Runnable() { public void run() { try { cl1.loadClass("A"); } catch(Exception e) {} }}).start();
try {
    cl2.loadClass("B");
} catch(Exception e) {};
....

public class ClassLoader1 extends ClassLoader {
    public ClassLoader1(URL[] urls) { super(urls); }
    public ClassLoader1(URL[] urls, ClassLoader p) { super(urls, p); }
    public Class<?> loadClass(String nm, boolean r) throws ClassNotFoundException {
        if (name.equals("A")) return cl2.loadClass(nm, r);
        else getParent().loadClass(nm, r);
    }
}
```

La conception des chargeurs (5)

la manipulation du byte code

- il est possible d'analyser puis modifier-générer le byte code avant l'appel à `defineClass()`
- 3 APIs principales : `bcel`, `asm`, `javassist`

les librairies et outils associés

- `bcel`
 - parser construisant une représentation arborescente de la classe (analogue à DOM pour XML)
 - représentation de tous les éléments (y compris les instructions de bas niveau)
 - possibilité de modifier la représentation (addition-suppression-modification des noeuds)
 - outil exigeant une connaissance approfondie des aspects "bas-niveaux" du langage
- `asm`
 - génération d'événements de parsing (analogue à SAX pour XML) traité par des listener
 - génération d'un événement pour tous les éléments (y compris les instructions de bas niveau)
 - possibilité de modifier la représentation (addition-suppression-modification des noeuds)
 - outil très puissant exigeant une connaissance approfondie des aspects « bas-niveaux) du langage
- `javassist`
 - construction d'une représentation arborescente de la classe (analogue à DOM pour XML)
 - représentation de tous les éléments (y compris les instructions de bas niveau)
 - possibilité de modifier la représentation (addition-suppression-modification des noeuds)
 - outil intégrant un compilateur n'exigeant PAS une connaissance approfondie des aspects "bas niveau" du langage

Une introduction à Javassist

la lecture-écriture du byte code

- Javassist utilise un cache et sa propre API de réflexion (puisque'une classe chargée ne peut plus être modifiée)

```
.....  
pool = ClassPool.getDefault();  
CtClass cc = pool.get(name);  
CtMethod[] meths = cc.getMethods();  
.....
```

le pool a le même CLASSPATH que le classloader courant. Il est possible de le modifier via les méthodes insertClassPath, appendClassPath

- Javassist a des méthodes d'écriture, de création d'une instance de Class et de génération du byte code

```
.....  
cc.writeFile();  
Class cls = cc.toClass();  
byte[] b = cc.toBytecode();
```

la modification du byte code

- les classes réflexives de Javassist ont des méthodes de transformation simple :
CtClass : setName, setSuperClass, addInterface, addConstructor, addField, removeInterface, removeConstructor,
CtMethod : insertBefore, insertAfter, insertAt, instrument, setBody, setWrappedBody

la modification des méthodes est très facile puisque les modifications sont indiquées via une String représentant le code java correspondant

- Javassist a aujourd'hui des limitations concernant les types generic et les annotations

La conception des chargeurs (6)

Un exemple simple de chargeur pour instrumenter les méthodes annotées par @instrument

```
public class SimpleInstrumentClassLoader extends ClassLoader {  
    private ClassPool pool;  
  
    public SimpleInstrumentClassLoader(ClassLoader parent, String cPath) throws Exception {  
        super(parent); pool = ClassPool.getDefault(); pool.insertClassPath(cPath);  
    }  
  
    protected Class<?> findClass(String name) throws ClassNotFoundException {  
        try {  
            CtClass cc = pool.get(name); //représentation réflexive Javassist de la classe  
            CtMethod[] meths = cc.getMethods(); //représentation réflexive Javassist des méthodes  
            for (int i = 0; i < meths.length; i++) {  
                Object[] annots = meths[i].getAnnotations(); //représentation réflexive Javassist des annotations  
                for (int j = 0; j < annots.length; j++) {  
                    Method mts = annots[j].getClass().getMethod("annotationType"); //obtention du type de l'annotation via la réflexion java  
                    if ((Class) mts.invoke(annots[j]).equals(classLoader.annotations.Instrument.class)) {  
                        String mthNme = meths[i].getName();  
                        String sBfr = "long __debut = System.currentTimeMillis(); System.out.println(\"debut de \" + mthNme + "\");";  
                        meths[i].insertBefore(sBfr);  
                        String sAfr = "long __duree = System.currentTimeMillis() - __debut; System.out.println(\"duree de \" + mthNme + \" = \" + __duree);";  
                        meths[i].insertAfter(sAfr);  
                    }  
                }  
            }  
        }  
        byte[] b = cc.toBytecode();  
        return defineClass(name, b, 0, b.length);  
    }  
    catch (Exception e) { throw new ClassNotFoundException(); }  
}
```

idem pour : **public** SimpleInstrumentClassLoader(String cPath)

les classes à instrumenter ont un CLASSPATH propre

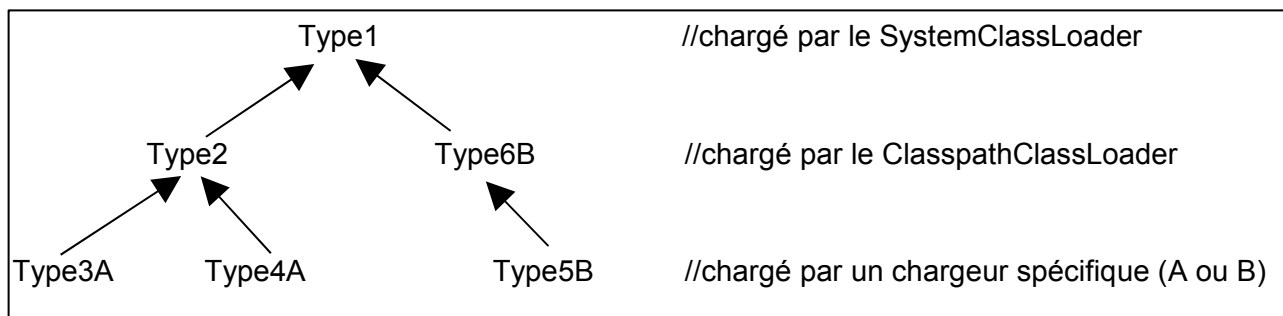
modification du code

génération du byte code

Les chargeurs et le typage (1)

La relation de sous-typage

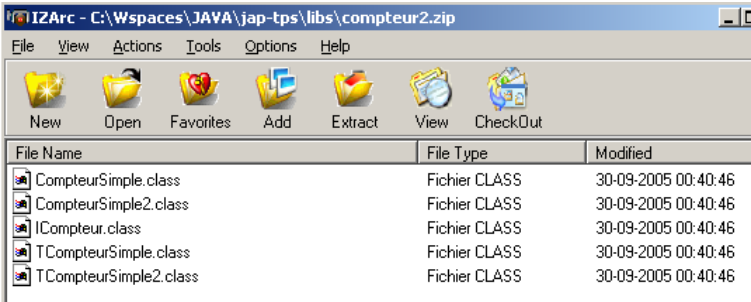
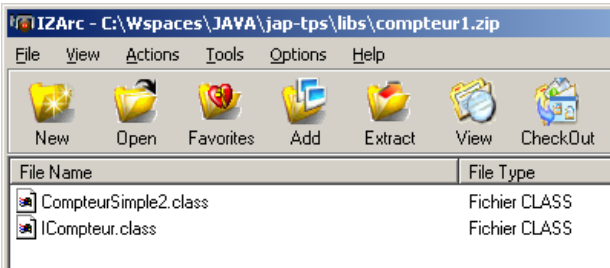
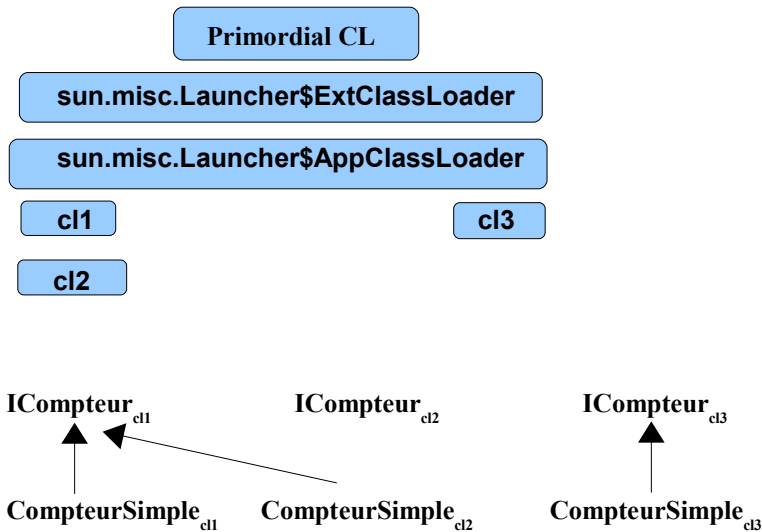
- le type prend en compte le ClassLoader
 - $[Type1, ClassLoader1] < [Type2, ClassLoader2] \iff Type1 < Type2 \text{ et } ClassLoader1 < ClassLoader2$
- le sous-typage dépend directement de la délégation entre les classloaders



les chargeurs et le typage (2)

un exemple (suite)

```
URL[] urls1 = new URL[] { new URL("file:///C:/Wspaces/JAVA/jap-tps/libs/compteur1.zip"); };
URL[] urls2 = new URL[] { new URL("file:///C:/Wspaces/JAVA/jap-tps/libs/compteur2.zip"); };
ClassLoader cl1 = new URLClassLoader1(urls1);
ClassLoader cl2 = new URLClassLoader1(urls2, cl1);
ClassLoader cl3 = new URLClassLoader1(urls2);
```



Attention la situation n'est pas symétrique :
Vue de cl2 ICompteur_{cl1} est un surtype de CompteurSimple_{cl2}
Vue de cl1 CompteurSimple_{cl2} n'est PAS st un soustype de ICompteur_{cl1}

L'utilisation d'un chargeur spécifique

une contrainte

- un ClassLoader obéissant au modèle java ne connaît que les classes chargées par lui ou ses parents
- un offre un ensemble étanche de classes

un exemple simple ...

- chargement via la méthode `loadClass(String className,boolean use)` → retourne une instance de `java.lang.Class`
- invocation via la réflexion d'une méthode de lancement

```
public class TestClassLoader2 {  
    public static void main(String[] args) {  
        URL url = null;  
  
        try {  
            url = new URL("file:///C:/Wspaces/JAVA/jap-examples/libs-classloader/philosophe0.jar");  
        } catch (MalformedURLException e) { e.printStackTrace(); }  
  
        URL[] urls = new URL[] { url };  
  
        URLClassLoader cl1 = new URLClassLoader(urls);  
        try {  
            Class cls= cl1.loadClass("philosophe0.LanceurPhiloActif");  
            Method meth = cls.getDeclaredMethod("main", new Class[] { String[].class });  
            meth.invoke(null, new Object[] { args });  
        } catch (Exception exc) { exc.printStackTrace(); }  
    }  
}
```

Le déchargement des classes

la spécification de java.....

- la JVM ne spécifie pas l'opération de déchargement considérée comme une optimisation dépendante de l'implémentation
- la règle généralement mise en oeuvre consiste à décharger les classe lorsque le ClassLoader responsable de leur chargement n'est plus référencé

Quelques généralités sur l'optimisation du chargement des classes

quelques règles de bon sens

- éviter le chargement des classes inutiles
- éviter une mauvaise gestion mémoire liée au chargement de classes
- réduire le temps d'obtention des .class

quelques solutions génériques

- utiliser de la réflexion
- regrouper des petites classes
- réduire la taille des .class

Eviter le chargement de classes inutiles

le principe

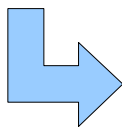
- la politique de chargement des classes diffère selon la JVM
 - elle est agressive dans certains cas : elle anticipe sur le besoin réel
- chargement dynamique des classes via l'API réflexive

ceci est de moins en moins vrai avec HotSpot

```
public interface IRegistry { ..... }  
public class PlainTextRegistry implements IRegistry { ..... }  
public class BinaryRegistry implements IRegistry { ..... }  
public class XMLRegistry implements IRegistry { ..... }  
public class Specific1Registry implements IRegistry { ..... }
```

```
public class RegistryFactory {  
    public static IRegistry getRegistry(String type) {  
        if (type.equals("plain")) return new PlainTextRegistry();  
        if (type.equals("binary")) return new BinaryRegistry();  
        if (type.equals("xml")) return new XMLRegistry();  
        if (type.equals("specific1")) return new Specific1Registry();  
    }  
}
```

toutes les classes sont chargées



```
public class RegistryFactory {  
    public static IRegistry getRegistry(String type) {  
        if (type.equals("plain")) return (IRegistry) Class.forName("PlainTextRegistry").newInstance();  
        if (type.equals("binary")) return (IRegistry) Class.forName("BinaryRegistry").newInstance();  
        if (type.equals("xml")) return (IRegistry) Class.forName("XMLRegistry").newInstance();  
        if (type.equals("specific1")) return (IRegistry) Class.forName("Specific1Registry").newInstance();  
    }  
}
```

seule la classe requise est chargée

Economiser la mémoire (1)

le principe

- la taille occupée en mémoire est généralement supérieure à celle du .class
- l'usage de très nombreuses petites classes peut conduire à une taille importante pour la zone des objets permanents
- une solution : le regroupement de plusieurs petites classes
 - cas des listeners (en particulier dans le cas du graphique)
 - cas des classes générées via JSP

Attention : il faut impérativement maintenir la cohérence sémantique !!



```
public class ControlPanel extends JPanel {
    public ControlPanel() {
        JButton open = new JButton("open");
        JButton close = new JButton("close");
        JButton save = new JButton("save");
        .....
        open.addActionListener(new OpenAction());
        close.addActionListener(new CloseAction());
        save.addActionListener(new SaveAction());
    }

    class OpenAction implements ActionListener {
        public void actionPerformed(ActionEvent e) { ..... }
    }
    class CloseAction implements ActionListener { ..... }
    class SaveAction implements ActionListener { ..... }
}
```

chaque inner class augmente la taille de ControlPanel d'environ 3K

```
public class ControlPanel extends JPanel {
    public ControlPanel() {
        JButton open = new JButton("open");
        JButton close = new JButton("close");
        JButton save = new JButton("save");
        .....
        ActionListener b = new ButtonAction();
        open.addActionListener(b);
        close.addActionListener(b);
        save.addActionListener(b);
    }

    class ButtonAction implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JButton b = (JButton) e.getSource();
            String cmd = b.getActionCommand();
            if (cmd.equals("open")) { .....
            } else if (cmd.equals("close")) {
            } else if (cmd.equals("save")) { ..... }
        }
    }
}
```


Economiser la mémoire (2)

l'utilisation de code "trampoline"

- limiter le nombre de classes de l'application via une classe de code "trampoline"

code d'aiguillage renvoyant
sur une autre classe

```
public class ControlPanel extends JPanel {
    public ControlPanel() {
        JButton open = new JButton("open");
        JButton close = new JButton("close");
        JButton save = new JButton("save");
        .....
        open.addActionListener(new ButtonListener(this, "open"));
        close.addActionListener(new ButtonListener(this, "close"));
        save.addActionListener(new ButtonListener(this, "save"));
    }

    public void open() { ..... }
    public void close() { ..... }
    public void save() { ..... }
}
```

```
public class ButtonListener implements ActionListener {
    private Object target;
    private Method method;

    public ButtonListener(Object tgt, String mth, Class ... args) {
        try {
            target = tgt;
            Class cls = target.getClass();
            method = cls.getMethod(mth, args);
        } catch (Exception e) { ..... }
    }

    public void actionPerformed(ActionEvent e) {
        try {
            method.invoke(target, new Object[] {} );
        } catch (Exception e) { ..... }
    }
}
```

perte de performances

la classe Proxy et l'interface InvocationHandler

- l'objectif est de disposer d'un code "trampoline" générique
- classe et interface du package java.lang.reflect
- la classe Proxy offre un code générique d'aiguillage vers un handler

Economiser la mémoire et réduire la durée de chargement

l'utilisation d'obfuscateurs

- outils pour complexifier la compréhension du code
- 3 grandes catégories d'algorithmes
 - layout obfuscation : renommage "confusionnant" des identificateurs et suppression des informations de debug
 - control obfuscation : complexification "apparente" du flot de contrôle via l'introduction de prédicats opaques (introduction de faux choix) et introduction de code contenant éventuellement des bugs
 - data obfuscation : répartition des structures de données du programme et cryptage des littéraux
modification des relations d'héritage, restructuration des tableaux (par linéarisation ou modification de la dimension), duplication de variables, ...
- de nombreux obfuscateurs : yguard, proguard, sandmark, jshrinker,
- réduction de la taille des .class : on peut demander à l'obfuscateur de réaliser uniquement un renommage des identificateurs en réduisant la taille des noms générés