CENTRE D'INNOVATION
EN TELECOMMUNICATION
& INTEGRATION DE SERVICES

# OSGi
# Partie II - Java

**Stéphane Frénot**

ARÈS



## Programmation Java

- Chargement de classes
  - Recherche de classes
  - Chargement dynamique
- Programmation orientée services
  - Framework
  - Contrat

## Chargement de classes

- Qui ?
- Où ?



## Les unités de compilation

- Le code source d'une classe est appelé *unité de compilation.*
- Une classe a un nom qui est composé du nom du package et du nom de la classe
  - java.util.Vector

# Les packages définition

- Unité d'organisation des classes
  - Organisation logique : `time.clock.Watch`
  - Organisation physique : `time/clock/Watch.class`
  - Il existe une correspondance directe entre le nom d'une classe et son emplacement dans le système de fichiers
- Espace de **nommage** hiérarchique
  - Description de la hiérarchie : `package time.clock;`
  - Notion de nom complet : `time.clock.Watch`
- Les bibliothèques java sont organisées en packages
  - java.util, java.net, org.objectweb...
  - Deux classes ayant le même nom complet ne peuvent pas s'exécuter en même temps ... en théorie



# Où ?

# Chargement de classe

```
Hello t;              /* La classe n'est pas encore chargée*/
t=new Hello();        /* Le classloader charge la classe en mémoire */
```

- Les outils du système cherchent toutes les classes
  - Compilation et Exécution : typage fort C & E
  - Les classes sont recherchées sur le système de fichiers
  - Les classes standards sont automatiquement trouvées par les outils
  - Pour indiquer l'endroit sur le système de fichiers à partir duquel il faut chercher les classes, on utilise le **classpath**



# Le classpath

- Il indique à partir de quel endroit rechercher une classe

```
java -classpath /usr/local/ titi.Toto
```

- La classe titi.Toto et **toutes** les classes lancées à partir de maintenant sont recherchées à partir du répertoire

```
/usr/local /*Résolution physique*/
```

- Il faut donc que la classe soit définie dans le fichier :

```
/usr/local/titi/Toto.class /*Résolution java*/
```

# Le classpath : le jar

- Un jar est une archive java
  - Elle permet le regroupement de fichiers dans un seul fichier
- Extension du système de fichiers

```
jar tvf toto.jar
```

```
tutu/
tutu/ours/
tutu/ours/Grumly.class
```

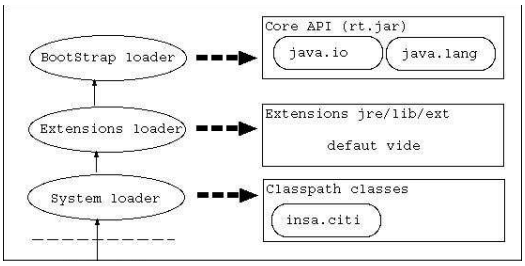
```
vi Test.java
```

```
package test;
import tutu.ours.Grumly;
public class Test {
  Grumly toto=new Grumly();
}
```

```
javac ? ???
```



# Chargement de classes




- Les classes sont recherchées par la machine virtuelle à trois emplacements
- Chaque emplacement est géré par un classloader
- Ces trois classloader sont définis au lancement de la machine virtuelle et sont immuables
- Ils sont organisés hiérarchiquement (avant de charger une classe, un Classloader demande à son parent de la charger)

# Chargement Implicite et Explicite

- Une classe est chargée implicitement par la machine virtuelle à lors de sa première instanciation

```
Hello t;            /* La classe n'est pas encore chargée*/
t=new Hello();      /* Le classloader charge la classe en mémoire */
```



# Chargement Explicite

- Le chargement d'une classe passe par les classloaders
- Il est possible de demander au classloader de charger une classe en se fondant sur la chaîne de caractères représentant son type
- Cette caractéristique permet de faire de la programmation semi-réflexive (programmation par les structures du langage)
- Il est possible de manipuler un type sans le posséder à la compilation, il n'est fournit qu'à l'exécution.

```
public static Class forName(String clazz);  /* Fabrique un type à partir de sa chaîne
                                               de description */
public Object newInstance(); /* Fabrique un objet conforme à la classe */
```

## Exemple manipulation explicite de type

```
package reflexion;

import java.lang.reflect.Method;
import java.util.Hashtable;

public class Test2 {
  public static void main(String [] arg) throws Exception{
    Class c=Class.forName("java.util.Vector");
    Object o=c.newInstance();
    java.util.Vector v=(java.util.Vector)o;
    v.add("coucou");
    System.out.println("vector"+v+": "+v.size()+" : "+v.get(0));

...
```



## Exemple manipulation explicite de type

```
...
    Object o2=c.newInstance();
    Method m1=c.getMethod("add", new Class[]{Object.class});
    m1.invoke(o2, new Object[]{new String("coucou")});
    Method m2=c.getMethod("size", null);
    System.out.println("Je suis un vector de "+m2.invoke(o2, null));
  }
}
```

## Le chargement explicite permet

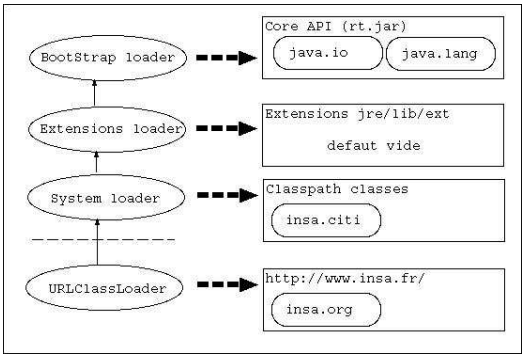
- de piloter finement le chargement d une classe java
- de piloter finement le chargement des ressources
- de gérer des versions d'une même classe

- *de Rendre un code client (utilisateur) totalement indépendant du code d'implantation de service*

- Interface ifc=ClassLoader.load("implementation");



## Extension des ClassLoader standards




==> Securité !
java -Djava.ext.dirs=myext MaClasse
grant codeBase "file:${java.home}/lib/ext/*"{permission java.security.AllPermission;};

## Une classe est donc définie par

- Un nom complet
  - test.toto.Titi
- Et un classloader qui la défini
  - URLClassloader
- Class c={nom, classloader}
- Si une classe n'est pas trouvée dans son classloader, on remonte dans la hiérarchie des classloaders



## Evolution de la programmation

- Nous sommes passé de la programmation
  - Client/serveur /* Programmation oo */
  - Client/Interface/Serveur /* Programmation Composant */
  - Client/Usine/Interface/Serveur /* Programmation service */
- Pour rendre un service de plus en plus adaptable
  - Mais de moins en moins simple à programmer

## Exemple : La classe Point
## Approche classique

```
public class Point {
  private int x;
  private int y;

  public void move (int dx, int dy){
    this.x+=dx;
  }
}
```

```
public class Client {
  public static void main( String [] arg){
    Point p=new Point(4,5);
    p.move(2,3);
  }
}
```

- Programmation OO
  - L'interface est liée à l'implantation
  - Il n'est pas possible de diffuser l'interface sans son implantation
  - Le cycle de vie du client est lié au cycle de vie du serveur



## Exemple : La classe Point
## Approche composant

```
public interface Point {
  public void move (int dx, int dy);
}
```

```
public class PointImpl implements Point {
  public void move (int dx, int dy){
    this.x+=dx;
  }
}
```

```
public class Client {
  public static void main(String [] arg){
    Point p=new PointImpl(4,5);
    p.move(2,3);
  }
}
```

- Ok pour la programmation Objet, mais…
  - Le client et le serveur sont liés par compilation
  - La classe d'implantaion du service est référencée dans le client...

## Dépendance client/serveur

```
public class Client {
  public static void main(String [] arg){
    Point p=new PointImpl(4,5);
    p.move(2,3);
  }
}
```

- La correction du serveur impose l'arrêt du client
- Client et serveur sont dépendants
- Pour les rendre indépendants il faut un conteneur/intercepteur/fabrique...
- Le modèle est Client/Conteneur/Serveur



## Programmation orientée service

- Un service est un composant
  - Interface de définition
  - Classe d'implantation
- Mais dont l'association Interface/Implantation est dynamique
  - Elle est fournie à l'exécution par un médiateur
  - Elle peut être remise en cause à tout moment
  - Elle isole entièrement le client du serveur **à l'interface de définition près**

# Un serveur de point

```
public class ServeurDeClassePoint{
  public static Point createClassPoint( ){
    URL[] urls=new URL[]{new URL("file:resource/")};
    ClassLoader cl=new URLClassLoader(urls);
    Point pt=(Point)cl.loadClass("tp1.PointImpl").newInstance();
    return pt;
  }
}
```

- Ne repose pas sur la classe, mais
- sur le nom de la classe (String)
- Il repose sur un « ClassLoader » qui charge la classe de définition
- ! A mettre en œuvre



# 2 améliorations classiques

```
package tp1;
public class ServeurDeClassPoint{
  static ClassLoader c1;
  static Class ptClass;
  static String url;
  static String className;

  public static Point createClassPoint(Point
tmp)
    throws Exception{
    if (ptClass==null){reloadImpl();}
    Point newPt=(Point)ptClass.newInstance();
    if (tmp!=null){
      newPt.initialize(tmp);
    }
    return newPt;
  }
```

```
public static void reloadImpl()
    throws Exception{
    setProperties();
    URL[] urls=new URL[]{new URL(url)};
    c1=new URLClassLoader(urls);
    ptClass=c1.loadClass(className);
  }
  private static void setProperties()
    throws Exception {
    URL u=new URL("file:params");
    BufferedReader is=new BufferedReader
      (new
        InputStreamReader(u.openStream()));
    url=is.readLine();
    className=is.readLine();
    System.out.println
      ("Les paramètres sont :
"+url+className);
  }
}
```

# Le client

```
package tp1;
public class Client{
  public static void main(String [] arg) throws Exception{
    while (true){
      Point p=ServeurDeClassPoint.createClassPoint(null);
      System.out.println(p);
      p.move(4,4);
      System.out.println(p);
      System.in.read();
      ServeurDeClassPoint.reloadImpl();
    }
  }
}
```



# Le client

- 1. Ne doit pas référencer l'implantation
- 2. Les versions d'implantation et d'interface sont les mêmes
- 3. Il faut pouvoir transmettre l'ancien état
- 4. Le client doit fournir la nouvelle et détruire l'ancienne référence

## La programmation orientée service

- Les services sont spécifiés par une interface et réalisés par une implantation
- Ils sont packagés dans le concept de bundle (composant = unité de déploiement = unité de packaging)
- Les clients n'instancient jamais l'implantation…Elle est fournie au run-time par un framework
- Le framework
  - Impose le modèle à composant utilisé
  - Gère le cycle de vie des composants
  - Gère les interactions entre composants
    - Appels directs/Notification



## OSGi platform