Cours 7 : Programmation répartie

- modèles de clients/serveur
- persistence
- sérialisation en Java, O'Caml et C
- chargement dynamique

Modèles de clients/serveur

- modèle orienté communication
  - protocole de communication
  - spécification des interactions
- modèle orienté traitement
  - conception d'une application conventionnelle
  - subdivision en modules exécutables sur plusieurs machines
- modèle orienté application
  - conception proche des applications centralisés
  - transparence des appels à distance :
    - RPC en C (NFS, ...)
    - RMI (en Java

Persistance

conservation d'un objet en dehors de l'exécution courante : dans un fichier ou un flux

But: récupération ultérieure dans le même programme ou un autre.

Difficultés: :

- lecture/écriture de données
- structures circulaires
- coercion de type à la relecture
- sauvegarde de code

Langages

- En Java : mécanisme de sérialisation d'objets!!!
- En O'Caml : sérialisation de valeurs, y compris fonctionnelles
- En C : représentation externe des données

Sérialisation en Java

- classes de flux
  - `ObjectOutputStream` : flux de sérialisation
  - `ObjectInputStream` : flux de désérialisation
- interface
  - `Serializable` : doit être implantée (vide) pour être sérialisé

Que stocke-t-on?

Le stockage contient :

- le nom et une clé (checksum) de la classe
- tous les champs (de données) sérialisables

La clé permet de vérifier la version de la classe.
Le modifieur `transient` permet d'indiquer qu'un champs ne doit pas être sérialisé.
possibilité de changer le mécanisme de sérialisaiton en implantant une classe `Externalizable` respectant les interfaces `writeExternal` et `readExternal`.

Lecture/Ecriture

- La méthode `void writeObject(Object)` écrit dans un flux (de type `ObjectOutputStream`) une instance de classe qui implante l'interface `Serializable`.
- La méthode `Object readObject()` lit dans un flux (de type `ObjectInputStream`) un persistant et retourne une valeur de type `Object`.
  - Une fois un `Object` obtenu, il est utile de faire une coercition de type (vers une sous-classe d'Object) sur la valeur lue pour pouvoir la manipuler avec son bon type.
  - Pour que cette contrainte de type puisse avoir lieu, il est nécessaire que la classe soit connue du programme (ou chargeable à ce moment là).
  - Si la classe n'existe pas une exception est alors déclenchée de type `ClassNotFound`.

Pas de transfert de code!

Exemple (1)

```
import java.io.*;
class Exemple20 implements Serializable   {
  String nom;
  Exemple20 autres;

  Exemple20() {nom=null;autres=null;}
  Exemple20(String n, Exemple20 e) {nom=n;autres=e;}

  boolean estVide() {return (nom == null);}
  void detruit() {nom=null;autres=null;}
  public String toString() {
    if (estVide()) return "[]";
    else if (autres.estVide()) return nom;
    else return nom+"::"+autres.toString();
  }
  public void ajoute (String unNom) {
    System.out.println("*");
    Exemple20 e = new Exemple20(nom,autres);
    autres=e;
    nom=unNom;
  }
}
```

Exemple (2)

```
class Execute  {
  public static void main (String[] args) {
    Exemple20 e = new Exemple20();
    ObjectOutputStream out;
    ObjectInputStream in;
    try {
      e.ajoute("machin");
      e.ajoute("truc");
      System.out.println("1 : "+e);
      out = new ObjectOutputStream(new FileOutputStream("Exemple20.dat"));
      out.writeObject(e); out.flush(); out.close();
      e.detruit();
      System.out.println("2 : "+e);
      in = new ObjectInputStream(new FileInputStream("Exemple20.dat"));
      e.ajoute("bidule");
      e.autres = (Exemple20)in.readObject();
      in.close();
      System.out.println("3 : "+e); }
    catch (java.lang.ClassNotFoundException exc){System.err.println(exc);}
    catch (StreamCorruptedException exc) {System.err.println(exc);}
    catch (IOException  exc) {System.err.println(exc);}
  }
```

Exécution

```
java Execute

*
*
1 : truc::machin
2 : []
*
3 : bidule::truc::machin
```

* : trace de `ajoute(..)`

Exécution commentée

La classe `Exemple20` est une classe pour les listes de chaînes de caractères. La représentation de la liste vide est une instance de cette classe dont les variables d'instance `nom` et `autres` valent `null`. Cette classe implante l'interface `Serializable`. Cette interface ne demande pas l'écriture de méthodes particulières.

- Les `*` indique l'ajout d'un élément à la liste.
- A la trace `1` il y a deux noms dans la liste. Celle-ci est sauvée dans le fichier `Exemple20.dat` puis détruite dans l'exécution du programme (trace 2).
- Une nouvelle tête de liste est créée.
- Le persistant stocké dans le fichier `Exemple20.dat` est ensuite relu et son type est forcé en `Exemple20` et ajouté en queue de liste. Le résultat est affiché à la trace 3.

On note que la contrainte de type effectue en fait un test dynamique de type sur la valeur qui vient d'être lue. Si la classe indiquée n'existe pas une exception `ClassNotFound` est alors déclenchée.

Threads persistants

Une instance d'une classe héritant de `Thread` et implantant `Serializable` peut être stockée à l'extérieur du programme.
Difficulté : arrêt et redémarage d'un thread
Exemple : Producteur/Consommateurs liés à un magasin
$\Rightarrow$ nécéssite de connecter le consommateur sur le magasin

Exemple (1)

```
class Product {

  String name;

  Product () {name = "Standard";}
  Product (String n) {name = n;}

  String getName() {return name;}

} // end class Product
```

Exemple (2)

```
class Shop {
  protected int size=8;
  protected Product [] buffer;
  protected int ip = 0;
  protected int ic = 0;


  Shop () { emptyShop();}

  Shop(int n){size=n;emptyShop();}

  void emptyShop() {
    buffer = new Product[size];
    for (int i=0; i< size; i++) {
      buffer[i]=new Product("Empty");
    }
  }

  void display1() {
    System.out.println("enter product : [("+(ip % size)+")]  " +
                       buffer[ip % size].getName());
  }
```

Exemple (3)

```
class Consumer extends Thread implements Serializable {
  transient Shop aShop;
  String name;

  Consumer(Shop s, String sn){aShop=s; name = sn;}

  void changeShop(Shop s) {
    aShop = s;
  }

  public void run() {
    while(true) {
        Product p = aShop.get();
        display(p);
        try {sleep((int)(1000*Math.random()));}
        catch (Exception e0) {}
   }
 }

  void display(Product p) {
        System.out.println("Get : "+p.getName()+ " from "+name);
  }
```

Exemple (4)

```
class Producer extends Thread {
  Shop aShop;
  int num = 0;
  Producer(Shop s) {aShop=s;}
  public void run() {
    while(true) {
      Product p = makeProduct();
      aShop.put(p);
      display(p);
      try {sleep((int)(333*Math.random()));}
      catch (Exception e0){}
    }
  }
  void display(Product p) {System.out.println("Put :"+p.getName()); }
  Product makeProduct() {
    Product p =  new Product("Product"+num);
    num++;
    return p;
  }
} //end class Producer
```

Exemple (5)

```
public class ProdConP {

  static void saveConsumer(Consumer c, String filename) {
    try { ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream(filename));
          out.writeObject(c);
          out.close();
    }
    catch (Exception e){ System.out.println(e);}
  }

  static Consumer loadConsumer(String filename) {
      Consumer c = null;
      try { ObjectInputStream in =
              new ObjectInputStream(new FileInputStream(filename));
              c = (Consumer)in.readObject();
              in.close();
      }
      catch (Exception e) { System.out.println(e);}
      return c;
  }
```

Exemple (6)

```
public static void main (String [] args) {
   Shop myShop = new Shop(12);
   Producer p = new Producer(myShop);   p. start();
   Consumer c1 = new Consumer(myShop,"Cl1");  c1.start();
   Consumer c2 = new Consumer(myShop,"Cl2");  c2.start();
   Consumer c3 = new Consumer(myShop,"Cl3");  c3.start();
   c1.stop();
   System.out.println("Arret de Cl1");
   char c;

   try {
     c=(char)System.in.read();
     switch (c) {
     case '1': { saveConsumer(c1,"C1.ser");break;}
     case '4': { Consumer cl = loadConsumer("C1.ser");
                 cl.changeShop(myShop);
                 cl.start();break;}
     }
   }
   catch (Exception e) {}
```

O'Caml : module Marshal

Persistants: non sûrs pour le typage

- `type extern_flag = No_sharing | Closure`
- `to_channel : out_channel -> 'a -> extern_flag list -> unit`
- `from_channel : in_channel -> 'a`
- `to_string : 'a -> extern_flag list -> string`
- `from_channel : string -> 'a`

```
let oc = open_out "toto.ser" in
  let r = Array.create 10 2.3 in
    to_channel oc r [] ; close_out oc;;

let ic = open_in "toto.ser" in
  let (r: float array) = from_channel ic in
    close_in ic; r;;
```

Danger

```
# let magic_copy a =
    let s = Marshal.to_string a [Marshal.Closures]
    in Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>
```

$\longrightarrow$ le bénéfice du typage statique est perdu :

```
#  (magic_copy 3 : float) +. 3.1;;
Segmentation fault
}}
```

Valeurs fonctionnelles (1)

Connaître les pointeurs de code :

- Réseau de machines compatibles
- Un même programme pour plusieurs machines

De nombreuses extensions // d'O'Caml utilise cette technique :

- BSML, OCAMLP3L, Caml-Flight, ...

Persistance avec `coca-ml`

Ide : Pour résoudre les problèmes de typage grâce au *cast* ăobjet dynamique.
Chaque classe hérite et est en relation de sous-typage avec `serialize` (pseudo classe ancêtre)
$\Rightarrow$ pour *downcast* vers sa classe d'origine un objet relu (ou d'une classe intermédiaire entre sa classe de construction et `serialize`).

Implantation

- définir un identificateur unique (ID) pour chaque classe : en créant un MD5 à partir de l'AST;
- trouver la Table des Méthodes (unique pour chaque classe)) : application partielle du constructeur;
- construire une table de hachage globale (ID, MT)
- sérialiser les structures circulaires;

nouveau module : `Dumpto`

- `to_string` : $\forall \alpha . \alpha \rightarrow string$
- `from_string` : $string \rightarrow serialize$

Exemple

```
# let l = new left 2;;
l : left = <obj>
# let o = let s = DumpTo.to_string l
   in  DumpTo.from_string  s;;
o : serialize = <obj>
# let t = cast o to top;;
t : top = <obj>
# let lt = cast o to left;;
lt : left = <obj>
# lt#eq(t);;
- : bool = true
# let r = cast o to right;;
Uncaugth exception : ...
```

Continuations

- contexte d’exécution + pile d’exécution

en C

- représentation des données différente selon les architectures
  - codage/décodage de chaque type pour toute architecture
  - représentation externe des données : XDR de SUN (RCF 1014)

    `http://www.faqs.org/rfcs/rfc1014.html`
    - format XDR
    - types XDR + fonction de codage/décodage

C : types XDR

- types de base : int, float, ...
- types structurés : tableaux (de longueur fixe), enregistrement, enumération, union

XDR codage : seules les données sont représentées pas leurs types

Flot XDR

- flot d'entrées/sorties

```
void xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op; XDR_ENCODE ou XDR_DECODE
```

- flot mémoire

```
void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

Cette routine initialise l'objet flux XDR pointé par `xdrs`. Les données du flux sont lues ou écrites dans le bloc mémoire situé en `addr` et dont la longueur ne dpasse pas `size` octets. L'argument op détermine la direction du flux XDR (`XDR_ENCODE`, `XDR_DECODE`, ou `XDR_FREE`).

Fonctions

```
xdr_bool(xdrs, bp)
      XDR *xdrs;
      bool_t *bp;


  xdr_string(xdrs, sp, maxsize)
      XDR
      *xdrs;
      char **sp;
      u_int maxsize;
```

Exemple

```
main(argc,argv)
char **argv;
{
struct hostent *h;
struct sockaddr_in
char buf[20];
int sock;
XDR xdr_handle1;
char mem [6];
int n = -1234;

...

xdrmen_create(&xdr_handle1, mem, 6, XDR_ENCODE);

...
}
```

Exemple (2)

```
if (!xdr_int(&xdr_handle1, &n)) {
  fprintf(stderr, "pb encodage entier\n");
  exit;
}
...
if (sendto(sock, mem, 6, 0, &sin, sizeof(sin)) == -1) {
  perror("pb sendto");
  exit();
}
```

Introspection

Objectifs: : explorer et de modifier des valeurs (des objets en Java) existant pendant une exécution.
Pour cela une classe elle-même est considérée comme un objet, ce qui permet :

- de connaitre la classe d'un objet et obtenir la description d'une classe;
- créer des objets sans connaître son nom à l'exécution;
- accéder et modifier les champs de données;
- invoquer une méthode

mais ne permet pas de définir une nouvelle classe ni ne modifier la hiérarchie des classes existante,

Motivations

- Il existe un véritable besoin de pouvoir explorer dynamiquement (sans connaître les sources de la compilation) l'intérieur des objets et des classes Java.
- Cela permet de construire (facilement) des extensions au langage et de pouvoir construire des environnements visuels (ne faisant qu'explorer de l'extérieur) comme par exemple pour les Java Beans.

API reflect

L'API java.lang.reflect permet, si les conditions de sécurité l'acceptent de :

- construire de nouvelles classes et des tableaux
- d'accéder et de modifier des champs d'objets existants
- d'invoquer des méthodes de classes ou d'objets
- d'accéder et de modifer des element des tableaux.

les domaines d'applications sont :

- accès de l'extérieur (JavaBeans);
- outils de développement, browser de classes, débogueurs, interprète, décompilateur.

classes pour l'introspection

- `java.lang`
  - `Object`
  - `Class`
- `java.reflect`
  - `Constructor`
  - `Field`
  - `Method`

classe Class

mt́hodes principales:

- static Class `forName`(String)
- `Object newInstance()`
- Field [] `Fields`()
- Method [] `Methods`()

Classe Class

- La classe `Class` représente une classe Java. On construit un objet `Class` soit en envoyant le message `getClass()` sur une instance, soit en la chargeant dynamiquement avec la méthode `Class.forName` auquel on passe le nom de la méthode sous forme de chaîne de caractères.
- On peut à partir d'une classe créer une nouvelle instance de celle-ci : la méthode `newInstance()` crée une nouvelle instance de la classe qui reçoit ce message en utilisant son constructeur sans paramètre.
- D'autre part il est possible de récupérer les informations d'une classe (champs, méthodes, constructeurs, `...`). On obtient alors des instances des classes : `Field`, `Method`, `Constructor` sous forme de tableaux. L'exemple suivant montre comment afficher ces informations.

Exemple d'exploration (1)

```
import java.lang.reflect.*;

public class Lecture {
  public static void main(String args[]) {
    Class c = null;
    Field[] champs = null;
    Method[] methodes = null;

    try {
      c = Class.forName(args[0]);
      champs = c.getDeclaredFields();
      methodes = c.getMethods();
    }
    catch (ClassNotFoundException e) { // ...;
                                       System.exit(0);}
    catch (SecurityException e) { // PB d'autorisation
                                    System.exit(0);}
```

Exemple d’exploration (1)

```
    for (int i=0; i< champs.length;i++) {
      Field uc = champs[i];
      System.out.println("champs "+i+" : "+uc);
    }

    for (int i = 0; i < methodes.length; i++) {
      Method um = methodes[i];
      System.out.println("methodes "+i+ " : " + um);
    }
  }

}
```

Exemple d'exploration (3)

Son exécution donne :

```
$ java Lecture java.lang.Boolean

champs 0 : public static final java.lang.Boolean java.lang.Boolean.TRUE
champs 1 : public static final java.lang.Boolean java.lang.Boolean.FALSE
champs 2 : public static final java.lang.Class java.lang.Boolean.TYPE
champs 3 : private boolean java.lang.Boolean.value
champs 4 : private static final long java.lang.Boolean.serialVersionUID
methodes 0 : public static java.lang.Boolean java.lang.Boolean.valueOf(jav
methodes 1 : public static boolean java.lang.Boolean.getBoolean(java.lang.
methodes 2 : public final native java.lang.Class java.lang.Object.getClass
methodes 3 : public int java.lang.Boolean.hashCode()
methodes 4 : public boolean java.lang.Boolean.equals(java.lang.Object)
methodes 5 : public java.lang.String java.lang.Boolean.toString()
methodes 6 : public final native void java.lang.Object.notify()
methodes 7 : public final native void java.lang.Object.notifyAll()
methodes 8 : public final native void java.lang.Object.wait(long)
             throws java.lang.InterruptedException
methodes 9 : public final void java.lang.Object.wait(long,int)
             throws java.lang.InterruptedException
methodes 10 : public final void java.lang.Object.wait()
```

création dynamique

Il est aussi possible de créer des instances de classes à partir de la méthode `newInstance` envoyée à une objet de type `Class`. Cette méthode retourne un `Object`. Une fois un objet crée, on récupère ses champs par `geField(name)`, et l'on peut accéder ou modifier la valeur de celui-ci (`get` et `set`).

Chargeur de classes utilisateur

- classe abstraite `java.lang.ClassLoader` :
  - charger le byte-code :
    méthode : `byte [] loadClassData (String)`
  - définir un objet `Class` à partie de cette suite d'octets :
    méthode : `Class defineClass(byte [], int,int)`
  - faire l'édition de liens :
    méthode : `resolveClass(Class)`
- la méthode `Class loadClass(String,bool)` effectue ces tâches.

Chargement

- La machine virtuelle Java charge dynamiquement les classes dont l'exécution du programme en cours a besoin. L'option `-verbose` de l'interprète de byte-code de la machine abstraite Java. Habituellement la machine virtuelle Java charge une classe à partir d'un fichier local. Ce chargement peut être dépendant du système (variable `CLASSPATH` sous Unix, ...). Néanmoins il peut avoir des situations où les classes doivent être chargées de manière différentes : classes distantes (accessibles à partir d'un serveur sur le réseau), format de fichier spécifique, conversion à la volée, modification de la sécurité. Pour ces cas, il est nécessaire de définir une sous-classe de la classe abstraite `ClassLoader` pour étendre le comportement de chargement.

Exemple (1)

L'exemple suivant, tiré du tutorial de Java, montre comment créer un chargeur de classes pour le réseau. La classe `NetworkClassLoader` définit deux méthodes : `loadClassData` qui d'une URL retourne un tableau d'octets correspondant au code transmis et `loadClass` (seule méthode abstraite de la classe `ClassLoader`) pour le chargement effectif. Elle contient d'autre part une table de hachage pour connaître les classes déjà transférées. `loadClass` vérifie si le nom de la méthode est déjà dans la table de hachage, si ce n'est pas le cas, elle transfère les données et construit la classe à partir d'un tableau d'octets, stocke la classe dans la table de hachage puis déclenche `resolveClass` pour autoriser la création d'instances.

Exemple (2)

```
class NetworkClassLoader extend ClassLoader {
    String host;
    int port;
    Hashtable cache = new Hashtable();
    private byte loadClassData(String name)[] {
    // load the class data from the connection
    ...
    }
    public synchronized Class loadClass(String name,
                                        boolean resolve) {
        Class c = cache.get(name);
        if (c == null) {
            byte data[] = loadClassData(name);
            c = defineClass(data, 0, data.length);
            cache.put(name, c);
        }
        if (resolve)
            resolveClass(c);
        return c;
    }
}
```

Exemple (3)

Le code suivant montre comment créer une instance de la classe `Main` chargée dynamiquement par le nouveau chargeur.

```
ClassLoader loader= new NetworkClassLoader(host,port);
Object main= loader.loadClass("Main", true).newInstance();
...
```

Les navigateurs WWW, intégrant une machine virtuelle Java, implantent une sous-classe de ClassLoader (abstraite) pour le transfert via le réseau des classes et pour modifier la sécurité (d'où un changement de comportement entre `appletviewer` et `netscape`).

Différents classLoaders

- Applet class loader : chaque navigateur en possede un (se basant sur l'URL CODEBASE)
- RMIclassLoader
- URLclassLoader : permet de charger des classes a' partir d'un ensemble d'URL

Exemple

```
try {
 urlList ul = {
  new URL ("http://www.infop6.jussieu.fr/classes"),
  new URL ("http://java.sun.com/myjar.jar")};
  ClassLoader lo = new URLClassLoader(urlList);
  Class c = loader.loadClass("MaClasse");
  MaClass mc = (MaClass)c.newInstance();
 }
...
```