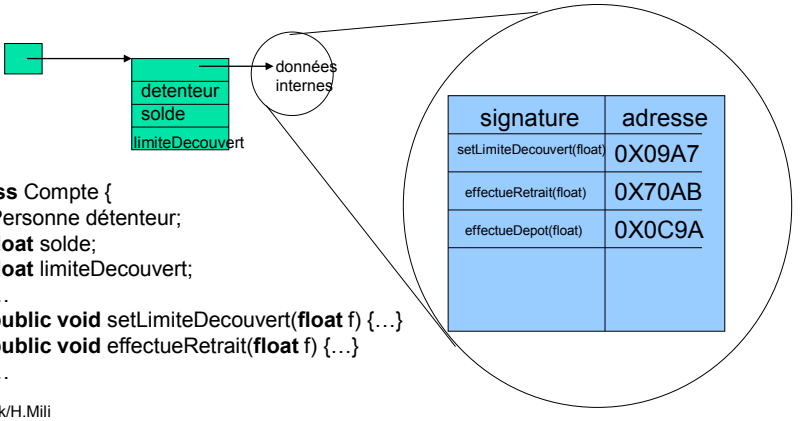
Sérialisation

- ***Introduction***
- Sérialisation : interface et classes
- Sérialisation et héritage
- Quelques problèmes
- Références

Sérialisation

La représentation des objets en mémoire durant l'exécution est complexe et spécifique à la JVM (pas trop)




```
class Compte {
   Personne détenteur;
   float solde;
   float limiteDecouvert;
   …
   public void setLimiteDecouvert(float f) {…}
   public void effectueRetrait(float f) {…}
   …
}
```

Sérialisation

- Ce format ne se prête ni à la sauvegarde:
  - Comment sauvegarder l'état d'un objet en dehors d'une session d'exécution, que l'objet soit sur la pile ou dans le heap
- Ni à l'échange:
  - La communication par les réseaux de télécommunications permet l'échange de messages sous forme de chaînes de caractères, mais rien de plus.
- Solution: "dumper" les objets sous forme de chaînes de caractères, en mettant des délimitateurs au bon endroit, pour pouvoir les reconstruire au besoin

Sérialisation

- En Java, les objets peuvent être "sérialisés" dans des Stream
  - **ObjectOutputStream** (pour les "dumper")
  - **ObjectInputStream** (pour les "charger")
- Ce ne sont pas tous les objets qui sont sérializable
  - Des raisons de sécurité
- Pour être sérialisable, un objet doit juste "le dire" (**implements** Serializable)

Interface Sérialisable

- Elle ne contient pas de méthodes
- C'est une "marker interface". Elle permet aux classes **ObjectOutputStream** et **ObjectInputStream** de savoir quoi faire
- Si on essaie de sérialiser un objet qui n'est pas sérialisable, on va avoir une exception
- Si l'objet que l'on sérialise contient une référence à un objet qui n'est pas sérialisable, on va aussi avoir un problème
- Ce qu'on a sérialisé, on va réussir à le désérialiser ☺

Classe ObjectOutputStream

- C'est une sous-classe de **OutputStream**
- Contient des méthodes spécialisées pour sérialiser certains types d'objets:
  - **public void** write(int data) **throws IOException**
  - **public void** write(byte[] b) **throws IOException**
  - **public void** writeBoolean(boolean data) **throws IOException**
  - **public final void** writeObject(Object obj) **throws IOException**

    Write the specified object to the ObjectOutputStream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written...
- La méthode writeObject(Object obj) procède de façon récursive
- Les classes Hashtable et Vector implémentent **Serializable** …

Exemple: Sérialisation

```
import java.io.*;
import java.util.*;
public class ObjetADisque {
 public static void main(String[] args) {
Message mess = new Message();
        String source  = "Jean Alain";
        String dest = "Abdel Obaid ";
        String[] lettre = {
                "Bonjour Alain",
                "Je te rappelle que nous avons",
                "une réunion ce lundi",
                "On se rencontre au SH-4321",
                "Si tu as besoin de documents",
                "N hésite pas a me contacter",
                "Bien à toi",
                "Abdel" };
        Date quand = new Date();
mess.writeMessage( source , dest, quand,
lettre);
try {    FileOutputStream fo = new
                        FileOutputStream("Fichier.obj");
        ObjectOutputStream oo = new
                        ObjectOutputStream(fo);
                oo.writeObject(mess);
                oo.close();
                System.out.println("C'est fait !");
  }  catch (IOException e) {;}
 }
}

class Message implements Serializable {
        int lignes;
        String De;
        String A;
        Date quand;
        String[] lettre;
        void writeMessage(
                String De_par, String A_par,
                Date quand_par,  String[] lettre_par)  {
                lettre = new String[lettre_par.length];
                for (int i =0 ; i < lettre_par.length; i++)
                        lettre[i]= lettre_par[i];
                lignes=lettre_par.length;
                A=A_par;  De=De_par;  quand=quand_par;
        }
}
```


Traitement spécial

- Les classes qui demandent une sérialisation spéciale doivent implémenter les deux méthodes suivantes:

  private void readObject(java.io.ObjectInputStream stream) throws IOException, ClassNotFoundException;

  private void writeObject(java.io.ObjectOutputStream stream) throws IOException

- La méthode ObjectOutputStream.writeObject(Object o) vérifie d'abord si la classe supporte ces méthodes. Si oui, elle les appelle.
- Pour faire une sérialisation par défaut à l'intérieur de writeObject(ObjectOutputStream), il faut appeler ObjectOutputStream.defaultWriteObject(). Elle ne sérialisera que la classe même (pas de récursion)

Sérialisation et héritage

- La sous-classe d'une classe sérialisable est sérialisable
- La sous-classe d'une classes non-sérialisable peut être sérialisable, mais il faudra s'occuper des champs hérités de la classe non-sérialisable
  - La superclasse doit avoir un constructeur sans arguments accessible à la sous-classe
  - En écrivant la méthode MaSousClasseSerializable.writeObject(..), je m'assure de sauvegarder les variables héritées
  - Je dois faire l'opération inverse dans la méthode readObject(ObjectInputStream) (je dois me souvenir dans quel ordre j'ai sauvegardé les champs hérités)
- On peut rendre la sous-classe d'une class sérialisable non-sérialisable en définissant les méthodes writeObject(…) et readObject(…) pour qu'elles lancent des exceptions

Désérialisation

- C'est l'opération inverse: on lit l'objet à partir du contenu d'un **ObjectInputStream**
- On peut ouvrir le **ObjectInputStream** sur un **FileInputStream** sur le même fichier dans lequel on a écrit l'objet
- Reconstruit un graphe d'objets hiérarchiquement (opération inverse de l'écriture)
- De nouveaux objets sont alloués (pour ne pas écraser ceux qu'on a utilisé pour sérialiser)

ObjectInputStream

- Des méthodes sont disponibles pour lire des objets de différents types:
  - public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException
  - public short readShort() throws IOException
  - public float readFloat() throws IOException
  - public void defaultReadObject() throws ClassNotFoundException, NotActiveException, IOException

    Read the non-static and non-transient fields of the current class from this stream. This may only be called from the readObject method of the class being deserialized. It will throw the NotActiveException if it is called otherwise.

- Si la classe de l'objet à reconstruire est chargée, OK, sinon, on essaie de la charger, et si on n'y arrive pas, ClassNotFoundException

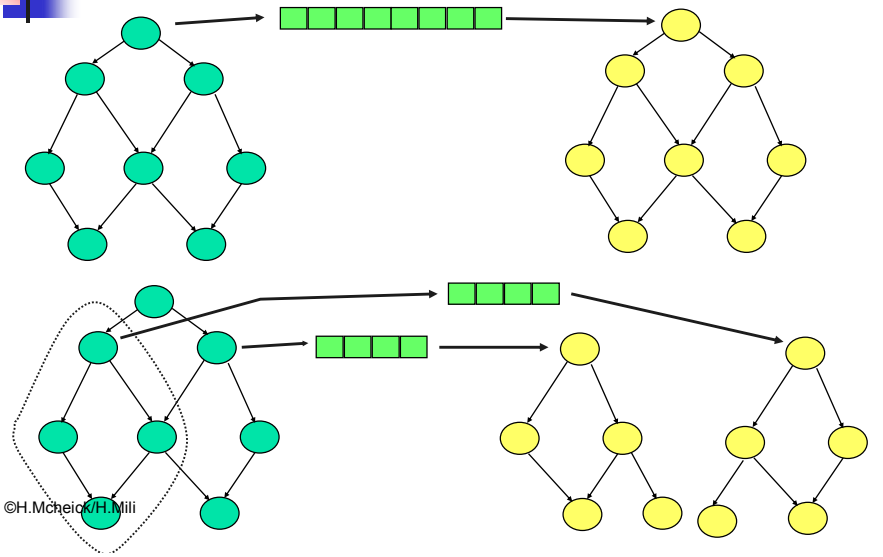
ObjectInputStream

- La séquence de reconstitution d'un objet sérialisé ressemble à l'appel du constructeur:
  - L'ordre de désérialisation est similaire à l'ordre d'appel des constructeurs
  - La désérialisation par défaut appelle le constructeur no-arg des (super) classes non-sérialisables, et restore les champs des ancêtres sérialisables en commençant par le haut
- Si on a sérialisé les champs appartenant à une superclasse non-sérialisable, on peut les restituer avec la méthode

```
private void readObject(java.io.ObjectInputStream stream) throws
   IOException, ClassNotFoundException;
```

- On peut reconstruire ces champs dans le même ordre qu'on les a sauvegardés
- On appelle ObjectInputStream.defaultReadObject()

Deux problèmes …



Problème de forward…

```
class Personne {
    String nom;
    Personne() {
        initialize();
    }
    public void initialize() {
        nom = "jean";
    }
}
class Client extends Personne {
    Vector commandes = new Vector();
    public void initialize() {
        String st = new String("xxx");
        commandes.add(st);
    }
}
```

```
public static void
main(String argv[]){
Client cl = new Client();
}
```

Problème de forward

- On ne pourra pas construire des objets du type Client!!!

- On retrouve ce problème lors de la désérialisation: cette dernière commence par l'appel du constructeur no-arg de la superclasse, qui peut appeler une méthode qui utilise une variable qui n'a pas encore été restituée!!!!

Échange d'objets avec Sockets

- Créer un *ObjectInputStream* avec le stream d'entrée de la socket
  ois = new ObjectInputStream(s.getInputStream());
- Créer un *ObjectOutputStream* avec le stream de sortie de la socket
  oos = new ObjectOutputStream(s.getOutputStream());

- Appeler la méthode writeObject() pour envoyer un objet
  oos.writeObject(new Date());
- Appeler la methode readObject pour recevoir un objet
  Date dateRecu = (Date) ois.readObject();

Serveur de date

```
import java.io.*;
import java.net.*;
import java.util.*;
public class DateServer {
   private ServerSocket dateServer;
   public static void main(String argv[ ]) throws Exception {
      new DateServer();
   }
   public DateServer() throws Exception {
      dateServer = new ServerSocket(3000);
      System.out.println("Server listening on port 3000.");
      this.start();
   }
   public void start() {
      while(true) {
         try {
            System.out.println("Waiting for connections.");
            Socket client = dateServer.accept();
            System.out.println("Accepted a connection from: "+
                                          client.getInetAddress());
            Connect c = new Connect(client);
         }
         catch(Exception e) {
         }
      }
   }
}
```

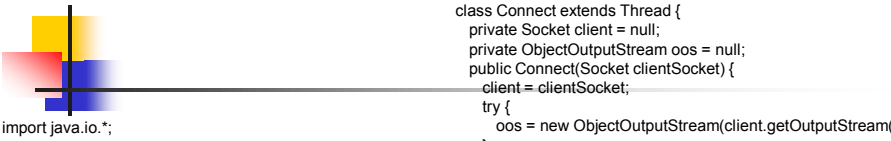


```
class Connect extends Thread {
   private Socket client = null;
   private ObjectInputStream ois = null;
   private ObjectOutputStream oos = null;
   public Connect(Socket clientSocket) {
      client = clientSocket;
      try {
         ois = new ObjectInputStream(client.getInputStream());
         oos = new ObjectOutputStream(client.getOutputStream());
      }
      catch(Exception e1) {
         try {
            client.close();
         }
         catch(Exception e) {
            System.out.println(e.getMessage());
         } return;
      } this.start();
   }
   public void run() {
      try {
         oos.writeObject(new Date());
         oos.flush();
         // close streams and connections
         ois.close();
         oos.close();
         client.close();
      }
      catch(Exception e) {
      }
   }
}
```

Client de date

```
import java.io.*;
import java.net.*;
import java.util.*;
public class DateClient {
   public static void main(String argv[ ]) {
      ObjectOutputStream oos = null;
      ObjectInputStream ois = null;
      Socket socket = null;
      Date date = null;
      try {
         // open a socket connection
         socket = new Socket("localhost", 3000);
         // open I/O streams for objects
         oos = new
ObjectOutputStream(socket.getOutputStream());
         ois = new
ObjectInputStream(socket.getInputStream());
         // read an object from the server
         date = (Date) ois.readObject();
         System.out.print("The date is: " + date);
         oos.close();
         ois.close();
      }
      catch(Exception e) {
         Sytem.out.println(e.getMessage());
      }
   }
}
```

Serveur de collection



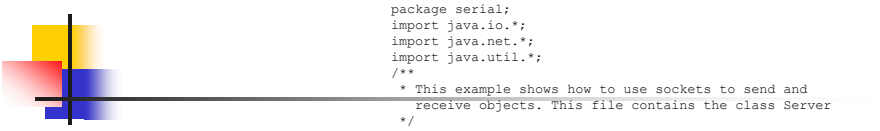
```
import java.io.*;
import java.net.*;
import java.util.*;
public class CollectionServer {
  private ServerSocket server;
  public static void main(String argv[ ]) throws Exception {
    new CollectionServer();
  }
  public CollectionServer() throws Exception {
    server = new ServerSocket(3000);
    System.out.println("Server listening on port 3000.");
    this.start();
  }
  public void start() {
    while(true) {
      try {
        System.out.println("Waiting for connections.");
        Socket client = server.accept();
        System.out.println("Accepted a connection from: "+
                                        client.getInetAddress());
        Connect c = new Connect(client);
      }
      catch(Exception e) {
      }
    }
  }
}
```

©H.Mcheick/H.Mili

```
class Connect extends Thread {
  private Socket client = null;
  private ObjectOutputStream oos = null;
  public Connect(Socket clientSocket) {
    client = clientSocket;
    try {
      oos = new ObjectOutputStream(client.getOutputStream());
    }
    catch(Exception e1) {
      try {
        client.close();
      }
      catch(Exception e) {
        System.out.println(e.getMessage());
      } return;
    } this.start();
  }
  public void run() {
    try {
Vector list = new Vector();
list.add(new UsagerInfo("charki", "noureddine", "ncharki", "uqam"));
list.add(new UsagerInfo("charki2", "noureddine2", "ncharki2","uqam2");
System.out.println("list sent...");
oos.writeObject(list);

      ois.close();
      oos.close();
      client.close();
    }
    catch(Exception e) {
    }
  }
}
```

Sérialisation avec client-serveur



```
package serial;
import java.io.*;
import java.net.*;
import java.util.*;
public class Client {
    public static void main(String args[]) {
        try {
          //1. Créer un nouveau Socket
          Socket soc=new Socket(
            InetAddress.getLocalHost(), 8020);
          //2.Obtenir  un  OutputStream  à  partir  du
      socket
          OutputStream o = soc.getOutputStream();
          //3.Créer un ObjectOutputStream à
          // partir du OutputStream
          ObjectOutput s = new ObjectOutputStream(o);
          //4. Ecrire les objets
          s.writeObject("Today");
          s.writeObject(new Date());
          //5. Fermer les Stream
          s.flush();
          s.close(); Exception e) {
          System.out.println(e.getMessage());
          System.out.println
        } catch ( ("Erreur");
          System.exit(1);
        }
    }
}
```

©H.Mcheick/H.Mili

```
package serial;
import java.io.*;
import java.net.*;
import java.util.*;
/**
 * This example shows how to use sockets to send and
   receive objects. This file contains the class Server
 */
public class Server {
    /**Create the serversocket and use its stream to
      receive serialized objects  */
    public static void main(String args[]) {
        ServerSocket ser = null;
        Socket soc = null;
        String str = null;
        Date d = null;
        try {
            //1. Créer un nouveau ServerSocket
            ser = new ServerSocket(8020);
            //1a. Obtenir un Socket à partir du ServerSocket
            soc = ser.accept();
            //2. Obtenir un InputStream à partir du socket
            InputStream o = soc.getInputStream();
            //3. Créer un ObjectInputStream à partir du InputStream
            ObjectInput s = new ObjectInputStream(o);
            //4. Lire les objets
            str = (String) s.readObject();
            d = (Date) s.readObject();
            //5. Afficher les objets
            System.out.println(str);
            System.out.println(d);
            //6. Fermer les Stream
            s.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.out.println("Erreur");
            System.exit(1);
        }
    }
}
```