Base de programmation Objet en JAVA. 3ème partie.

Frédéric Vernier
(Université Paris-Sud / LRI / LIMSI-CNRS)
Frederic.Vernier,@limsi.fr

Ce cours reprend en grande partie le matériel pédagogique mis au point par Jérôme Nobécourt, Christian Jacquemin et Claude Barras pour l'enseignement de Java en 2001 et 2002 en FIIFO et celui trouvé sur Internet
(http://www.laltruiste.com/, etc.)

Rappel important

- Ne pas confondre classe et instance
  - On décrit les classe dans le langage JAVA
  - On créé et on tue les instances pendant le déroulement du programme
  - Le déroulement du programme est décrit dans les fonctions des classes (en commençant par main() de votre objet principal)
    - Fonctions statiques = appelées depuis n'importe où, sans contexte spécifique
    - Fonction d'instance = appelée comme un service demandé à une instance

Rappel important

- Instance
  - Comme un autre programme indépendant
  - Capable de gérer son état personnel (données)
  - Capable de rendre un service contextuel sans que l'appelant se soucie de l'état de l'instance
- Classe
  - Stocke des données et des fonctions uniques
  - Fournit un service général
  - Sert de modèle a la création d'instance

Instance

- Décrite au travers de la classe qui lui servira de modèle
- Créée n'importe où grâce au mot-clé "new"
  - Depuis une méthode statique de classe
  - Depuis la méthode d'une instance
- N'est accessible qu'au travers de références que les autres (classes et instances) du programme ont de cette instance
  - Le "new" renvoie cette référence
  - Cette référence se manipule comme un int ou un float (affectation, paramètre d'une méthode)

Référence

- Les références sont la seule façon de collaborer avec d'autres instance (sauf soi-même si on est une instance)
- Lorsque une instance n'a plus de références sur elle-même, elle est détruite car inutile
- Une référence est typée par
  - La classe …
  - … qui a servit a créé l'instance
  - … sur laquelle on stocke une référence
- Mais ne pas confondre référence et classe !!!

Rappel important

- Une classe hérite une partie de sa description d'une description existante (une autre classe)
  - Pour éviter les redites
  - Pour que les services d'une instances soient organisés en couches (des services spécifiques aux plus génériques)
  - Pour ne pas faire d'hypothèses trop strictes sur la classe exacte de l'instance avec laquelle on travail
  - Mélanger des instances de classes différentes dans une structure de donnée
    - Une liste d'objets à dessiner (Ronds, Rectangles, etc.)
    - Une hiérarchie de Fichier/Répertoire

Héritage

- `Public class Fils extends Pere`
  - Le fils récupère les membres de son père
  - Les Fils peut se servir des membres protected de son père
  - Les étrangers peuvent se servir des membres publiques hérités du père

Interface

- `Public class Fils extends Pere implements Metier`
- On a envie que les étrangers puissent se servir d'un même service dans plusieurs classes
- On décrit les fonctions qui correspondent à ce service dans un fichier à part
- On dit explicitement qu'une classe remplit les exigence de l'interface

Référence

- Une référence à une instance peut être
  - Du type exact de la classe qui a servit à la créée
  - Du type de la classe du père (plus générique)
    - On ne peut pas appeler les membres spécifiques
    - On peut transtyper pour avoir une ref. spécifique
  - Du type d'une des interfaces que remplit la classe
    - On peut appeler une fonction de l'interface sans savoir du tout de quelle classe il s'agit
    - On peut toujours transtyper
- Quand au déroulement d'une fonction
  - On ne sait pas qui nous appelle (sauf si argument)
  - On ne sait pas ce que l'appelant sait de nous

Classe abstraite

- Mélange de père et d'interface
  - Le fils hérite des fonctions non abstraites
  - Le fils doit implémenter les fonctions abstraites
- On ne peut pas instancier de classes abstraites ou d'interfaces car ce sont pas des descriptions concrètes

Plan 5: Exception et Entrées-sorties

1. Introduction
2. Rattrapage d'exception
3. Syntaxe try{… }catch(…){…}
4. Factorisation de catch(s)
5. Exemple
6. Exemple type de Fichier
7. Classe Exception
8. Affichage d'exception
9. Jeter ses propres exceptions
10. Exemple CreateNewFile()
11. Signature et Héritage
12. Conseils de Christian
13. Conseils de Bruce
14. Entrées-Sorties
15. 0bjets d'Entrées-Sorties
16. Chaine d'Entrées-Sorties
17. Entrées-Sorties de char
18. Entrées-Sorties d'objets
19. Sérialisation
20. Utilisation
21. Objets divers d'Entrées-Sorties
22. System.xxx
23. Exemple : écrire une Exception

Introduction

- Quelques axiomes de base en informatique :
  - Un programme sans bugs, ça n'existe pas
  - Un utilisateur qui ne fait aucune erreur, ça n'existe pas
  - Un matériel qui marche toujours comme il devrait, ça n'existe pas
  - Une erreur d'exécution peut arriver n'importe quand. Exemple : " plus de mémoire "
- Un logiciel doit pouvoir gérer "un minimum" ces quatre points.
  - En Java, le mécanisme de tolérance aux fautes est réalisé à travers les exceptions

Rattrapage

- Un comportement anormal en cours d'exécution provoque une exception
- Une exception est un message. Il est retourné à la méthode qui a appelé le traitement erroné
- Une exception est créée
  - soit par la JVM (erreur interne, erreur de conception)
  - soit par l'instruction throw du programmeur
- Les instructions try/catch permettent de surveiller un bloc d'instructions
- Principe : La méthode qu reçoit une exception la transmet à la méthode qui l'a appelée et ainsi de suite mais on voudrait pas tout planter !

Syntaxe

```
try {
  blocASurveiller;
}
catch (NomClasseException NomClasseException_Arg) {
  traitement;
  if ()
    NomClasseException_Arg.printStackTrace();
}
[finally {dansTousLesCas}]
```

- Les instructions dans la clause *catch* ou finally ne sont pas surveillées par le mécanisme
- Si on attrape une exception elle ne se propage plus au dessus (ouf !)

Factorisation de catch

- On peut traiter plusieurs exceptions dans un seul try/catch

```
public class LectInt {
    public static void main(String[] args) {
   byte tab[]=new byte[10];

   try {System.out.println( "\nSaisie d'un entier : ");
        System.in.read( tab );
        String uneChaine=(new String(tab)).trim();

        int unEntier=Integer.parseInt(uneChaine);
        System.out.println( "\nEntier saisi : " + unEntier);

        boolean unBooleen=tab[12]==tab[0];
        System.out.println( "\nBooléen saisi : " + unBooleen);

   } catch( NumberFormatException NumberFormatException_Arg) {
        System.err.println( "Ce n'est pas un entier !");
   } catch( Exception Exception_Arg ) {
         Exception_Arg.NomClasseException_Arg.printStackTrace();
   } finally {
        System.err.println( "Fin du try/catch");
   }
}
```

Exemple

- Exception sur un accès incorrect à un tableau.

```
> java LectInt
Saisie d'un entier :
32
Entier saisi : 32
java.lang.ArrayIndexOutOfBoundsException ...
Fin du try/catch
Fin d'exécution
```

- Exception sur une conversion incorrecte.

```
> java LectInt
Saisie d'un entier :
2er4
Ce n'est pas un entier!
Fin du try/catch
Fin d'exécution
```

Exemple type

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

La classe Exception

- Les exceptions sont des classes java dont la classe mère est Throwable
  - Error : erreur interne de la JVM => On ne doit rien faire
  - RuntimeException : erreurs de conception => pas de catch (NullPointerException) mais allez debuguer !
  - Les autres : il ***faut*** un catch
  - Le codeur pose des conditions sur l'appel de sa fonction
  - La fonction appelante DOIT traiter le cas dans lequel …
- On peut faire ses propres exceptions avec des sous-classes de Exception
- Catch (Exception) gère toutes exceptions

Affichage

- Par défaut si une exception n'est pas attrapée elle plante le programme et s'affiche comme suit :

```
java.lang.StringIndexOutOfBoundsException: String index out of range: -46
  at java.lang.String.substring(Unknown Source)
  at FTDebug.debug(FTDebug.java:475)
  at FTPresentation.mousePressed(FTPresentation.java:301)
  at java.awt.AWTEventMulticaster.mousePressed(Unknown Source)
  at java.awt.Component.processMouseEvent(Unknown Source)
  at java.awt.Component.processEvent(Unknown Source)
  at java.awt.Component.dispatchEvent(Unknown Source)
  …
  at FTProg.main(FTProg: 54)
```

- On peut forcer l'affichage dans un catch()

```
catch(MyException MyException_Arg) {
   MyException_Arg.printStackTrace();
}
```

Jeter ses propres exceptions

```
Class MonException extends Exception {
  MonException(){}
  MonException(String s){super(s);}
}
```

- Utilisation : throw new MonException();
- Une méthode qui peut déclencher des exceptions (throw) DOIT indiquer dans sa signature la liste de ces exceptions (throw**s**)

```
nomMethode(arg) throws Exception1, Exception2 {
throw (new Exception1("mon erreur s'est produite"));
```

- Si vous ne pouvez pas gérer l'exception relancez la au dessus !

```
catch(Exception exception_Arg) {
   System.err.println("An exception was thrown");
   throw exception_Arg; }
```

Exemple : File.createNewFile()

public boolean createNewFile()
throws IOExceptionAtomically

creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file. This method, in combination with the deleteOnExit() method, can therefore serve as the basis for a simple but reliable cooperative file-locking protocol.

Returns:
true if the named file does not exist and was successfully created; false if the named file already exists

Throws:
IOException - If an I/O error occurred
SecurityException - If a security manager exists and its SecurityManager.checkWrite(java.lang.String) method denies write access to the file

Since:
1.2

Exemple : createNewFile()

```
Public File creerTemp() throws IOException {
  File File1 = new File ("temp.txt");
  try {
    File1.createNewFile();
  } catch (SecurityException SE_Arg) {
    File1 = new File ("/tmp/temp.txt");
    File1.createNewFile();
  }
  return File1;
}
```

Signature et Héritage

- Si m() est une méthode de A,
- Si B est une sous-classe de A
  - La surcharge d'une méthode m() dans B ne pourra pas émettre des exceptions qui n'ont pas été initialement prévues dans la méthode m() de A
- Si vous créez B pour remplacer A, B doit s'utiliser de la même façon
- Les exceptions possibles font parties de la signature
- Java cherche l'exception la plus proche du type qu'elle doit traiter
  - Si rien ne correspond ou si pas de try on remonte
  - Le premier truc qui correspond est utilisé et tout s'arrête !

Conseils de Christian

- Quand faut-il gérer des exceptions ?
  - Obligatoire si vous utilisez une méthode avec une signature contenant throws
- Exemple: la méthode read()
  - Pour les I/O
  - Pour gérer vos propres erreurs
  - Pour les messages d'erreurs
- Quand ne faut-il pas le faire ?
  - Quand l'erreur se produit très très rarement
  - Quand un if est suffisant pour le faire
  - Quand la méthode "au-dessus" sait le faire
  - Quand vous ne savez pas quoi mettre dans le catch

Conseils de Bruce

- Gérer le problème au niveau adéquate. (éviter d'attraper l'exception quand on ne sait pas quoi en faire)
- Résoudre le problème et appeler la méthode qui a plantée à nouveau
- Mettre une rustine et continuer sans appeler la fonction incriminée
- Calculer un résultat alternatif (approximation) a la place de ce qui était censé être calculé
- Faire son possible pour pas tout planter puis relancer la même exception aux plus haut appelants en espérant qu'ils puissent finir la réparation
- Pareil que point précèdent mais lancer une exception d'un autre type pour indiquer e qui reste a faire
- Sortir proprement du programme (fermer les connexions réseau, etc.)
- Simplifier. Si vos exception rendent les choses plus compliquées elle seront douloureuses a utiliser
- Rendez vos librairies et vos application plus surs, c'est un investissement a court terme pour le debugging et a long terme pour la robustesse

Entrées-Sorties

- Les mécanismes objets permettent d'organiser les liens à l'intérieur du programme
- On a besoin de liens avec le monde extérieur (disque dur, réseau, utilisateur, etc.)
- Souvent sous forme de flux de données
- Clavier, fichiers, pipes…
- paquetage java.io centralise toutes les classes et interfaces
- Les erreurs d'entrées/sorties (très fréquentes) sont des exceptions sous IOException
- 4 classes abstraites:
  - InputStream : flux d'octets en entrée
  - OutputStream: flux d'octets en sortie
  - Reader : flux de caractères en entrée
  - Writer : flux de caractères en sortie
- 1 classe pour les fichiers : File

Objets d'Entrées-Sorties

- Pour lire/écrire dans un fichier en utilisant seulement son nom, il faut associer un flux d'octets sur fichier
  - FileInputStream pour la lecture
  - FileOutputStream pour l'écriture

```
FileInputStream fluxFichLect = new FileInputStream(path+nom);
```

- Pour lire/écrire plus efficacement, on utilise les versions "bufferisées" (bloc par bloc plutôt que octet par octet) :
  - BufferedInputStream pour les lectures
  - BufferedOutputStream pour les écritures

```
BufferedInputStream tampLect=new BufferedInputStream(fluxFichLect, 32);
```

- Pour lire/écrire des données de type simple (int, float, etc.):
  - DataInputStream pour les lectures
  - DataOutputStream pour les écritures

```
DataInputStream tamponLectureTypeX=new DataInputStream (tampLect);
int i=tamponLectureTypeX.readInt();
```

Chaîne d'entrées-sorties

- Les entrées–sorties se connectent en chaîne

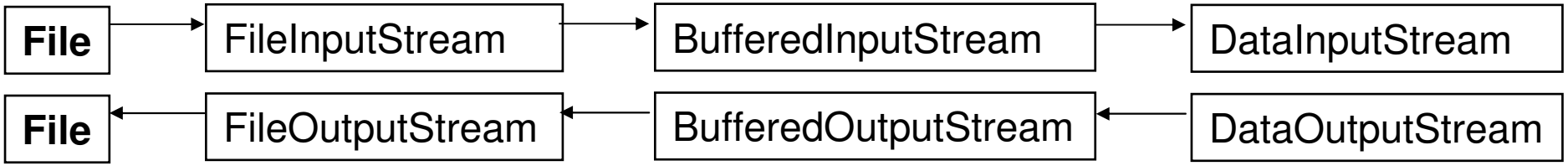
```
FileOutputStream fluFichEcr = new FileOutputStream("SaveObj.bin");
BufferedOutputStream tampEcr = new BufferedOutputStream(fluFichEcr, 32);
DataOutputStream tamponEcrType = new DataOutputStream (tampEcr);
tamponEcrType.writeInt(1249);
tamponEcrType.flush();
fluFichEcr.close();

FileInputStream fluFichLec = new FileInputStream("SaveObj.bin");
BufferedInputStream tampLect = new BufferedInputStream(fluFichLec, 32);
DataInputStream tamponLectureTypeX = new DataInputStream (tampLect);
int i=tamponLectureTypeX.readInt();
fluFichLec.close();

System.out.println(i);
```

Attention au try-catch

- La compilation impose IOException



Entrées-Sorties de caractère

- Pour les écritures: Pour lire/écrire dans un fichier, il faut associer un flux (stream) à une instance de File.
  - FileReader pour les entrées
  - FileWriter pour les sorties

```
FileReader fluxFichLecture= new FileReader(new File(path,nom));
```

- Pour lire/écrire plus efficacement, on utilise des flux " bufferisés " associé à un flux:
  - BufferedReader pour les lectures readline()
  - BufferedWriter pour les écritures write();newLine()

```
BufferedReader tampLect=new BufferedReader(fluxFichLecture);
```

- PrintWriter permet d'écrire facilement des fichiers textes.

```
PrintWriter fichierTexte=new PrintWriter(tamponEcriture);
fichierTexte.println(uneChaine);
```

Entrées-Sorties d'Objets

- Pour lire/écrire des objets (instances)
  - ObjectInputStream

```
try {
  FileInputStream istream = new FileInputStream("t.tmp");
  ObjectInputStream p     = new
   ObjectInputStream(istream);
  int i          = p.readInt();
  String today = (String)p.readObject();
  Date date      = (Date)p.readObject();
  istream.close();
} catch (FileNotFoundException
   FileNotFoundException_Arg){…}
  catch (IOException IOException_Arg) {…}
  catch (ClassNotFoundExceptionIOException_Arg) {…}
```

Attention à l'ordre

• FileNotFound… est une IOException, pas ClassNotFound !

  - ObjectOutputStream: writeLong(long val);writeObject(Object obj)

Sérialisation

- Ce que nous venons de voir
  - Est lié à la programmation orientée objet
  - S'appelle la sérialisation !
- Ne marche pas entre deux versions successives d'un objet (même modif. mineure)
- Les classes des instances sérialisées / désérialisées doivent implémenter
  - Serializable (rien à faire)
  - Externalizable (2 petites fcts à implémenter)

Utilisation

- Fichier
  - Sauvegarde (reprise sur panne, persistance)
  - Fichier propriétaire, binaire
- Réseau
  - Partage d'instances entre deux clients
  - Création d'instances sur serveur et distribution
- Base de données
  - Stockage d'instances dans une BD
  - Recherche

Objets divers d'Entrées-Sorties

- Les PipedOutputStream et PipedInputStream permettent de connecter des entrées avec des sorties
- RandomAccessFile : read, write et seek()
- StringReader (et writer) lit une String plutôt qu'un fichier texte
- PushbackInputStream permet de revenir en arrière (pour suivant)
- System.in, System.out, et System.err dont des flux : on va pouvoir jouer avec !

system.xxx

- System.in est un InputStream brut

```
import java.io.*;

public class Echo {
  public static void main(String[] args)
  throws IOException {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    String s;
    while((s = in.readLine())!=null && s.length()!=0)
      System.out.println(s);
    // An empty line or Ctrl-Z terminates the program
  }
}
```

Attention aux throws

- pas besoin de try-catch

Attention : double chaînage en 1 ligne

Attention à s = in.readLine())!=null

- stockage (dans s) et test en même temps

Exemple : écrire une Exception

```
// créer le fichier de sortie
try {
  File1                    = new File("traces.txt");
  FileWriter1              = new FileWriter(File1, true);
}
catch (IOException IOException_Arg) {IOException_Arg.printStackTrace();}

// jette une exception gratuite et la récupère
try {throw(new Exception());}
catch (Exception Exception_Arg) {
  StringWriter StringWriter = new StringWriter();
  PrintWriter  PrintWriter  = new PrintWriter(StringWriter);
  Exception_Arg.printStackTrace(PrintWriter);
  StringBuffer Error        = StringWriter.getBuffer();

  StringTokenizer ST1 = new StringTokenizer(Error.toString(), "\n\t");

  // les 2 premières lignes ne sont pas intéressantes
  ST1.nextElement();
  ST1.nextElement();

  while (ST1.hasMoreElements()) {FileWriter1.write(ST1.nextToken());}
}
```