**Classes utilitaires Java**

|  |  |
| --- | --- |
| Chaine de caractère  teste si c'est un nombre |  |
| /\*\*  \* isNumeric <br>  \* <p>  \* Méthode utilitaire déterminant si une chaine de caractère contient une valeur numérique  \* </p>  \*  \* @param strNum chaine de caractère à tester  \* @return true si la chaine contient un nombre entier ou décimal relatif, false sinon  \*/  public static boolean isNumeric(String strNum) {  if(strNum != null) {  return strNum.matches("-?\\d+([\\.\\d+)](\\\\.\\\\d+))?");  }  return false;  } | Classe métier |
| teste si une chaine contient au moins un caractère non blanc |  |
| /\*\*  \* isEmptyOrBlank <br>  \* <p>  \* Méthode utilitaire permettant de déterminer si une chaine de caractère est nulle ou ne contient que des caractères blancs  \* </p>  \*  \* @param s chaine à tester  \* @return true si nulle, vide ou ne contenant que des caractères blancs, fasle sinon  \*/  public static boolean isEmptyOrBlank(String s) {  return s == null || s.trim().isEmpty();  } | Classe métier |
| extrait la première occurrence d'un pattern s'il est trouvé |  |
| /\*\*  \* extrairePremiereOccurrence <br>  \* <p>  \* Methode utilitaire permettant d'extraire d'une chaine de caractère le premier élément correspondant au pattern  \* </p>  \*  \* @param pattern  \* à extraire  \* @param source  \* chaine de caractère  \* @return valeur extraite correspondant au pattern si trouvée, null sinon  \*/  public static String extrairePremiereOccurrence(Pattern pattern, String source) {  Matcher matcher = pattern.matcher(source);  if (matcher.find()) {  return matcher.group();  }  return null;  } | Classe métier |
| extrait toutes les occurrences d'un pattern |  |
| /\*\*  \* extraireToutesOccurrences <br>  \* <p>  \* Methode utilitaire permettant d'extraire tous les éléments correspondant au pattern d'une chaine de caractère  \* et les retourne dans une liste  \* </p>  \*  \* @param pattern  \* à extraire  \* @param source  \* chaine de caractère  \* @return la liste des valeurs extraites et correspondant au pattern  \*/  public static List<String> extraireToutesOccurrences(Pattern pattern, String source) {  List<String> liste = new ArrayList<>();  Matcher matcher = pattern.matcher(source);  while (matcher.find()) {  liste.add(matcher.group());  }  return liste;  } | Classe métier |
|  |  |
| Listes  teste si le contenu de 2 listes est identique sans tenir compte de l'ordre |  |
| /\*\*  \* sameContent<br>  \* <p>  \* Méthode utilitaire permettant de comparer le contenu de deux collections sans tenir compte de  \* l'ordre des éléments  \* </p>  \*  \* @param collA  \* @param collB  \* @return true si tous les éléments sont présents dans les deux listes, false sinon  \*/  public static <T> boolean sameContent(Collection<T> collA, Collection<T> collB) {  if(Objects.nonNull(collA) && Objects.nonNull(collB)) {  return collA.size() == collB.size() && collA.containsAll(collB) && collB.containsAll(collA);  }  return Objects.isNull(collA) && Objects.isNull(collB);  } | Classe métier |
| teste si les éléments d'une collection sont incrémentés d'un certain pas à partir d'une valeur initiale |  |
| /\*\*  \* incremented <br>  \* <p>  \* Méthode utilitaire permettant de déterminer si une liste de nombre est bien incrémentée de N  \* en N à partir de la valeur initiale passée en paramètre  \* </p>  \*  \* @param coll collection de valeurs à tester  \* @param step valeur du pas d'incrémentation  \* @param initialValue valeur initiale attendu  \* @return true si la valeur minimale correspond à celle passée en paramètres et que tous les  \* intermédiaires de N en N jusqu'à la valeur maximale sont présents dans la liste,  \* false sinon  \*/  public static boolean incremented(Collection<Integer> coll, Integer step, Integer initialValue) {  Integer somme = coll.stream().reduce(0, (x, y) -> x + y);  Integer sommeRef = (int) ((2 \* initialValue + (coll.size() - 1) \* step) \* coll.size() \* 0.5);  return Integer.compare(somme, sommeRef) == 0;  } | Classe métier |
|  |  |
| **Interfaces**  interface permettant de définir des TriFonctions |  |
| @FunctionalInterface  public interface TriFunction<A, B, C, R> {  R apply(A a, B b, C c);  default <V> TriFunction<A, B, C, V> andThen(Function<? super R, ? extends V> after) {  Objects.requireNonNull(after);  return (A a, B b, C c) -> after.apply(apply(a, b, c));  }  } | Classe Model |
|  |  |
| Incrémentation  incrémente facilement une série de caractère pouvant être des chiffres, des lettres ou un mélange des 2 |  |
| /\*\*  \* Increment <br>  \* <p>  \* Classe permettant d'incrémenter facilement une séquence de chiffre et ou de  \* lettre  \* </p>  \*  \* @author a-ramade  \* @since 09/2022  \*  \*/  public class Increment {  /\*\* Compteur \*/  private int count = 1;  /\*\* Séquence incrémentée \*/  private char[] chars;  /\*\* Difini si tous les caractères ont le même type \*/  private IncrementType uniqueType;  /\*\* Option d'ajout de caractère si séquence limite atteinte \*/  private boolean addNewChar = false;  /\*\* Active l'incrémentation après avoir restitué la première valeur \*/  private boolean activeIncrement = false;  /\*\* Défini les types et les limites des caractères incrémentables \*/  private enum IncrementType {  DIGIT('0', '9'), LOWERCASE('a', 'z'), UPPERCASE('A', 'Z');  /\*\* première valeur de la séquence \*/  private char start;  /\*\* dernière valeur possible de la séquence \*/  private char end;  private IncrementType(char start, char end) {  this.start = start;  this.end = end;  }  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* CONSTRUCTEUR \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* Constructeur par défaut<br>  \* <p>  \* Créé une instance de type Increment permettant d'incrémenter facilement une  \* séquence de caractère  \* </p>  \*  \* @param firstValue  \* valeur de départ  \*/  public Increment(String firstValue) {  super();  this.initIncrement(firstValue);  }  /\*\*  \* onstructeur avec option<br>  \* <p>  \* Créé une instance de type Increment permettant d'incrémenter facilement une  \* séquence de caractère<br>  \* L'option addNewChar permet d'ajouter un nouveau caractère UNIQUEMENT si la  \* séquence contient des caractères de même type  \* </p>  \* <div>  \* <ul>  \* <li>Exemple: valeur = 'ZZ'</li>  \* <li>valeur suivante avec addNewChar = false : 'AA'</li>  \* <li>valeur suivante avec addNewChar = true : 'AAA'</li>  \* </ul>  \* </div>  \*  \* @param firstValue  \* valeur de départ  \* @param addNewChar  \* active ou non l'ajout d'un nouveau caractère lorsque la séquence  \* ateint sa plus grande valeur possible  \*/  public Increment(String firstValue, boolean addNewChar) {  super();  this.initIncrement(firstValue);  }  /\*\*  \* initIncrement<br>  \* <p>  \* Méthode d'initialisation : découpe la première valeur de la séquence en  \* tableau de caractères  \* </p>  \*  \* @param firstValue  \* première valeur de la séquence  \*/  public void initIncrement(String firstValue) {  if (firstValue != null && firstValue.trim().length() > 0) {  this.chars = firstValue.toCharArray();  }  if (firstValue.matches("\\d+")) {  this.uniqueType = IncrementType.DIGIT;  } else if (firstValue.matches("[A-Z]+")) {  this.uniqueType = IncrementType.UPPERCASE;  } else if (firstValue.matches("[a-z]+")) {  this.uniqueType = IncrementType.LOWERCASE;  }  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* METHODES \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  /\* \*\*\*\*\* \*\*\*\*\* METHODES D'APPEL \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* increment<br>  \* <p>  \* Méthode permettant d'obtenir la séquence incrémenté d'une unité  \* </p>  \*  \* @return la séquence incrémentée d'une unité  \*/  public String increment() {  if (activeIncrement) {  this.count++;  this.chars = incrementChars(chars, chars.length - 1);  } else {  activeIncrement = !activeIncrement;  }  return String.valueOf(chars);  }  /\*\*  \* increment<br>  \* <p>  \* Méthode permettant d'obtenir une chaine de caractère incluant la séquence  \* incrémenté d'une unité <br>  \* Format de la chaine : prefix-sequenceincrémentée-sufixe.extension  \* </p>  \*  \* @param prefix  \* @param prefixSeparator  \* @param suffixSeparator  \* @param suffix  \* @param extension  \* @return chaine contenant la séquence incrémentée  \*/  public String increment(String prefix, String prefixSeparator, String suffixSeparator, String suffix,  String extension) {  StringBuilder result = new StringBuilder(prefix).append(prefixSeparator);  if (activeIncrement) {  this.count++;  this.chars = incrementChars(chars, chars.length - 1);  } else {  activeIncrement = !activeIncrement;  }  return result.append(chars).append(suffixSeparator).append(suffix).append(extension).toString();  }  /\* \*\*\*\*\* \*\*\*\*\* METHODES DE TRAITEMENT \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* incrementChars<br>  \* <p>  \* Méthode permetant d'incrémenter la chaine de caractère  \* </p>  \* <div>  \* <ul>  \* <li>teste le type du caractère en cours</li>  \* <li>si le caractère a atteint la limite de sa séquence, réinitialise le  \* caractère et incrémente le caractère supérieur par récursivité</li>  \* <li>sinon, incrémente le caractère d'une unité</li>  \* <li>si tous les caractères de la séquence ont ateint leur limite ET qu'ils  \* sont tous de même type ET que l'option est activée, alors ajoute un caractère  \* suplémentaire au début de la séquence</li>  \* </ul>  \* </div>  \*  \* @param chars  \* tableau de caractères à incrémenter  \* @param index  \* du caractère à traiter  \*/  private char[] incrementChars(char[] chars, int index) {  if (index > -1) {  IncrementType incrementType = Character.isDigit(chars[index]) ? IncrementType.DIGIT  : Character.isUpperCase(chars[index]) ? IncrementType.UPPERCASE : IncrementType.LOWERCASE;  if (chars[index] == incrementType.end) {  chars[index] = incrementType.start;  return incrementChars(chars, index - 1);  } else {  ++chars[index];  }  } else if (addNewChar && Objects.nonNull(uniqueType)) {  chars = new char[chars.length + 1];  for (int i = 0; i < chars.length; i++) {  chars[i] = uniqueType.start;  }  if (uniqueType == IncrementType.DIGIT) {  chars[0] = '1';  }  }  return chars;  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* GETTEURS / SETTEURS \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  public int getCount() {  return count;  }  public String getCurrentValue() {  return String.valueOf(chars);  }  } | Classe métier |
|  |  |
| Lecteur de fichier  lit une liste de fichiers ligne par ligne et interprète les données selon la méthode passée en paramètre |  |
| *Exemple d'utilisation : fichiers CSV avec les anciens tirages du Loto*  *// interprétateur*  public static Function<String, Tirage> extraireTirage = (csvValue) -> {  String[] datas = csvValue.split(CSV\_SEPARATOR);  if (datas.length == 2) {  LocalDate date = LocalDate.parse(datas[0], datas[0].matches(DATE\_FORMAT\_STD) ? DATE\_FORMATER\_STD : DATE\_FORMATER\_SP);  return new Tirage(date, datas[1], datas[2], datas[3], datas[4], datas[5]);  }  return null;  };  *// méthode faisant appel au lecteur de fichiers*  public static List<Tirage> extraireAnciensTirrages() {  List<Tirage> tirages = LecteurFichier.readAndInterpretLinesInFiles(PATH\_FICHIER, NOMS\_FICHIERS\_STD, CSV\_LINE\_PATTERN\_STD, extraireTirage);  tirages.addAll(LecteurFichier.readAndInterpretLinesInFiles(PATH\_FICHIER, NOMS\_FICHIERS\_SP, CSV\_LINE\_PATTERN\_SP, extraireTirage));  return tirages;  } |  |
| /\*\*  \* LecteurFichiers <br>  \* <p>  \* Charge le contenu d'un ou plusieurs fichiers, extrait de chaque ligne un  \* motif spécifique avant d'instancier une liste d'objet à partir de ces données  \* </p>  \*  \* @author a-ramade  \* @since 09/2022  \*  \*/  public class LecteurFichier {  /\*\*  \* readAndInterpretLinesInFiles <br>  \* <p>  \* Charge le contenu d'un ou plusieurs fichier et recherche/extrait de son  \* contenu la valeur spécifiée  \* </p>  \*  \* @param <R> type d'objet à instancier à partir des données des fichiers  \* @param path adresse des fichiers  \* @param files noms des fichiers  \* @param patternToExtract motif à extraire dans chaque ligne  \* @param interpreter fonction instanciant un objet à partir des données issues de chaque ligne  \* @return une liste d'objet de type <R>  \*  \*/  public static <R> List<R> readAndInterpretLinesInFiles(String path, String[] files, String patternToExtract,  Function<String, R> interpreter) {  Pattern pattern = Pattern.compile(patternToExtract);  List<R> list = new ArrayList<>();  for (String nomFichier : files) {  try (Stream<String> sourceLines = Files.lines(Paths.get(path, nomFichier))) {  list.addAll(sourceLines.map(line -> extractFirstOccurence(pattern, line)).filter(Objects::nonNull).map(interpreter).filter(Objects::nonNull).collect(Collectors.toList()));  } catch (IOException e) {  e.printStackTrace();  }  }  return list;  }  /\*\*  \* extractFirstOccurence <br>  \* <p>  \* Methode utilitaire permettant d'extraire d'une chaine de caractère le premier  \* élément correspondant au pattern  \* </p>  \*  \* @param pattern à extraire  \* @param source chaine de caractère  \* @return valeur extraite correspondant au pattern si trouvée, null sinon  \*/  private static String extractFirstOccurence(Pattern pattern, String source) {  Matcher matcher = pattern.matcher(source);  if (matcher.find()) {  return matcher.group();  }  return null;  }  } | Classe métier |
|  |  |
|  |  |

**Outils Java**

Outils permettant de faciliter le dévellopement

|  |  |
| --- | --- |
| Tests  classes permettant de réaliser facilement des tests pour valider une Regex, le formatage d'une date, etc… |  |
| public class Catacteres {  private final static String strTest = "5 001,55 €";  private final static String ESPACE\_CLASSIQUE = " ";  private final static String ESPACE\_INSECABLE = "\u00A0";  public static void testerEspaceInsecable() {  System.out.println(String.format("Classique \t %s", Character.codePointAt(ESPACE\_CLASSIQUE, 0)));  System.out.println(String.format("Insecable \t %s", Character.codePointAt(ESPACE\_INSACABLE, 0)));  System.out.println(String.format("Apostrophe \t %s", Character.codePointAt("'", 0)));  System.out.println("-----------------------------");  for (int i = 0, l = strTest.length(); i < l; i++) {  System.out  .println(String.format("%s \t %s", strTest.substring(i, i + 1), Character.codePointAt(strTest, i)));  }  }  } | pack test |
| public class Date {  public static void test() {  /\*  \* LocalDateTime ld = LocalDateTime.parse("2002-12-17T14:10:55");  \*  \* System.out.println(ld.plus(50, ChronoUnit.DAYS));  \*  \* System.out.println(ld.getDayOfWeek());  \* System.out.println(ld.getDayOfMonth());  \* System.out.println(ld.getDayOfYear()); System.out.println(ld.getMonth());  \* System.out.println(ld.getMonthValue()); System.out.println(ld.getYear());  \* //System.out.println(ld.getEra()); System.out.println(ld.toString());  \*  \* System.out.println("---------------------------------------------");  \*  \* Period p = Period.between(ld.toLocalDate(), LocalDate.now());  \* System.out.println(p.getYears()); System.out.println(p);  \*/  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yy HH'h'mm'min'ss's'");  LocalDateTime ldt = LocalDateTime.parse("01/12/19 14h10min05s", formatter);  DateTimeFormatter testFormat = DateTimeFormatter.ofPattern("S");  System.out.println(ldt.format(testFormat));  }  } |  |
| public class RegEx {  private final static String REGEX = ".\*\\S+.\*";  private final static Pattern PATTERN = Pattern.compile(REGEX);  public static void testerRegex() {  List<String> tests = new ArrayList<>();  tests.add("");  tests.add(" ");  tests.add(" ");  tests.add("user1@domain.com");  tests.add("4");  tests.add("4.1");  tests.add("IRSA Titre 4");  tests.add("IRSA Titre 3.2");  tests.add("mdjf5.2sgs");  tests.add("mdjf 6.3 sgs");  tests.add("mdjf 6.3 sgs 7");  for (String test : tests) {  Matcher matcher = PATTERN.matcher(test);  System.out.println(test + " => " + matcher.matches());  }  tests.stream().forEach(System.out::println);  }  } |  |
|  |  |
|  |  |
|  |  |
| Traitement de String et de liste de String  réaliser facilement des séquences de tout type de traitements sur une liste de String |  |
| /\*\*  \* TraiterListeStrings <br>  \* <p>  \* Classe utilitaire permettant de réaliser de nombreux traitements de bases sur  \* une liste de chaines de caractères  \* </p>  \*  \* @author a-ramade  \* @since 2022  \*  \*/  public class TraiterListeStrings {  private Stream<String> liste;  private static final String SEPARATEUR = "\\r\\n";  private String LISTE\_REF = "CODEPLI\_212\r\n" + "CODEPLI\_0230\r\n" + "CODEPLI\_0231\r\n" + "CODEPLI\_0211\r\n" + "CODEPLI\_0211\r\n" + "CODEPLI\_0223\r\n" + "CODEPLI\_0224\r\n" + "CODEPLI\_0228\r\n" + "CODEPLI\_0225\r\n" + "CODEPLI\_0227\r\n" + "CODEPLI\_0228\r\n" + "CODEPLI\_0232\r\n" + "CODEPLI\_11";    /\*\* \*\*\*\*\* \*\*\*\*\* CONSTRUCTEURS \*\*\*\*\* \*\*\*\*\* \*/    public TraiterListeStrings() {  extraireListe();  }    public TraiterListeStrings(List<String> listeStrings) {  liste = listeStrings.stream();  }    public TraiterListeStrings(String onlyOneString) {  liste = Arrays.asList(onlyOneString).stream();  }    public TraiterListeStrings(String stringADecouper, String separateur) {  liste = Arrays.asList(stringADecouper.split(separateur)).stream();  }  private void extraireListe() {  liste = Arrays.asList(LISTE\_REF.split(SEPARATEUR)).stream();  }    /\*\* \*\*\*\*\* \*\*\*\*\* METHODES UTILITAIRES \*\*\*\*\* \*\*\*\*\* \*/    private static String extraireValeur(Pattern pattern, String source) {  Matcher matcher = pattern.matcher(source);  if (matcher.find()) {  return matcher.group();  }  return "";  }  /\*\* \*\*\*\*\* \*\*\*\*\* METHODES DE TRAITEMENT \*\*\*\*\* \*\*\*\*\* \*/    public TraiterListeStrings ajouterGuillemets() {  liste = liste.map(str -> String.format("\"%s\"", str));  return this;  }  public TraiterListeStrings ajouterMotifs(String prefixe, String sufixe) {  liste = liste.map(str -> String.format("%s%s%s", prefixe, str, sufixe));  return this;  }  public TraiterListeStrings remplacerMotif(String motifARemplacer, String elementAInserer) {  Pattern pattern = Pattern.compile(motifARemplacer);  liste = liste.map(str -> pattern.matcher(str).replaceAll(elementAInserer));  return this;  }  /\*\*  \* extraireMotif<br>  \* <p>  \* extrait un motif dans chaque élément de la liste<br>  \* Exemple d'utilisation :<br>  \* liste = DV\_NUMDECL=1234, DV\_NUMSINI=5678, DV\_CODNATU=BDG  \* extraireMotif("DV\_[A-Z]{7}") => DV\_NUMDECL, DV\_NUMSINI, DV\_CODNATU  \* </p>  \*  \* @param motifAExtraire  \* @return l'instance de TraiterListeStrings  \*/  public TraiterListeStrings extraireMotif(String motifAExtraire) {  Pattern pattern = Pattern.compile(motifAExtraire);  liste = liste.map(str -> {  Matcher matcher = pattern.matcher(str);  if (matcher.find()) {  return matcher.group();  }  return null;  }).filter(Objects::nonNull);  return this;  }  /\*\*  \* extraireVariable<br>  \* <p>  \* extrait une valeur variable au sein d'un motif constant<br>  \* Exemple d'utilisation :<br>  \* liste = action: imprimer, action: sauvegarder, action: restaurer<br>  \* extraireVariable("action: (.\*)") => imprimer, sauvegarder, restaurer  \* </p>  \* <u>utiliser le motif (.\*) ou (.\*?) pour identifier la partie variable à  \* extraire</u>  \*  \* @param motifAvecVaribleAExtraire  \* @return l'instance de TraiterListeStrings  \*/  public TraiterListeStrings extraireVariable(String motifAvecVaribleAExtraire) {  String motif = String.format(".\*%s.\*", motifAvecVaribleAExtraire);  Pattern pattern = Pattern.compile(motifAvecVaribleAExtraire);  liste = liste.filter(str -> str.matches(motif)).map(str -> {  Matcher matcher = pattern.matcher(str);  if (matcher.find()) {  return matcher.group(1);  }  return null;  });  return this;  }    public TraiterListeStrings ordonnerListe() {  liste = liste.sorted();  return this;  }    public TraiterListeStrings ordonnerListe(String motifOrdonnancement) {  Pattern pattern = Pattern.compile(motifOrdonnancement);  liste = liste.sorted((str1, str2) -> extraireValeur(pattern, str1).compareTo(extraireValeur(pattern, str2)));  return this;  }  public TraiterListeStrings appliquerStringFormatPattern(String stringFormatPattern) {  int occurrence = 0;  Matcher m = Pattern.compile("%s").matcher(stringFormatPattern);  while (m.find()) {  occurrence++;  }  if (occurrence == 1) {  liste = liste.map(str -> String.format(stringFormatPattern, str));  } else if (occurrence == 2) {  liste = liste.map(str -> String.format(stringFormatPattern, str, str));  } else if (occurrence == 3) {  liste = liste.map(str -> String.format(stringFormatPattern, str, str, str));  }  return this;  }  public TraiterListeStrings supprimerDoublons() {  liste = liste.distinct();  return this;  }  public TraiterListeStrings supprimerEspaces() {  liste = liste.map(str -> str.trim());  return this;  }    /\*\* \*\*\*\*\* \*\*\*\*\* AFFICHAGE DU RESULTAT \*\*\*\*\* \*\*\*\*\* \*/    public TraiterListeStrings afficherTailleListePuis() {  List<String> listeCourrante = liste.collect(Collectors.toList());  System.out.println(String.format("Nombre d'éléments dans la liste = %d", listeCourrante.size()));  liste = listeCourrante.stream();  return this;  }  public void afficherTailleListe() {  System.out.println(String.format("Nombre d'éléments dans la liste = %d", liste.count()));  }    public void afficherListe() {  liste.forEach(System.out::println);  }  public void afficherEnLigne(String separateur) {  System.out.println(liste.collect(Collectors.joining(separateur)));  }  } | pack traitementstringetliste |
| compare le contenu de 2 listes de String |  |
| /\*\*  \* ListesSimples <br>  \* <p>  \* Outils permettant de comparer et d'effectuer des traitements simples sur 2  \* listes de chaines de caractères  \* </p>  \*  \* @author a-ramade  \* @since 2021  \*  \*/  public class ListesSimples {    private static List<String> listeRef;  private static List<String> listeTest;  private final static String SEPARATEUR = "\r\n";  private final static String LISTE\_REF = "A\r\n" + "B\r\n" + "C\r\n" + "B";  private final static String LISTE\_TEST = "A\r\n" + "B\r\n" + "D\r\n";    /\*\* \*\*\*\*\* \*\*\*\*\* CONSTRUCTEURS \*\*\*\*\* \*\*\*\*\* \*/    public ListesSimples() {  listeRef = Arrays.asList(LISTE\_REF.split(SEPARATEUR));  listeTest = Arrays.asList(LISTE\_TEST.split(SEPARATEUR));  }  public ListesSimples(List<String> listeRef, List<String> listeTest) {  ListesSimples.listeRef = listeRef;  ListesSimples.listeTest = listeTest;  }    /\*\* \*\*\*\*\* \*\*\*\*\* METHODES DE TRAITEMENT \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* comparerContenus <br>  \* <p>  \* Compare les éléments contenus dans les 2 listes (sans tenir compte de  \* l'ordre) et affiche les différences  \* </p>  \*/  public static void comparerContenus() {  String manquantDansListeTest = listeRef.stream().filter(code -> !listeTest.contains(code))  .collect(Collectors.joining(SEPARATEUR));  System.out.println(String.format("Elements attendus et non présents dans la liste TEST : %s", manquantDansListeTest));  String enTropDansListeTest = listeTest.stream().filter(code -> !listeRef.contains(code))  .collect(Collectors.joining(SEPARATEUR));  System.out.println(String.format("Elements en trop non présents dans REF : %s", enTropDansListeTest));  }  /\*\*  \* testerDoublons <br>  \* <p>  \* Vérifie pour chacune des listes si elle contient des éléments en doublon et  \* affiche ces éléments le cas échéant  \* </p>  \*/  public static void testerDoublons() {  Set<String> set = new HashSet<>();  System.out.println(  String.format("Elements en doublons dans la liste REF : %s", listeRef  .stream().filter(str -> !set.add(str)).collect(Collectors.joining(", "))));  set.clear();  System.out.println(String.format("Elements en doublons dans la liste TEST : %s",  listeTest.stream().filter(str -> !set.add(str))  .collect(Collectors.joining(", "))));  }    /\*\*  \* mettreEnLigne <br>  \* <p>  \* Affiche le contenu des liste en une seule ligne  \* </p>  \*  \*/  public static void mettreEnLigne() {  System.out.println("Liste REF = " + mettreEnLigne(LISTE\_REF));  System.out.println("Liste TEST = " + mettreEnLigne(LISTE\_TEST));  }  private static String mettreEnLigne(String chainePrincipale) {  String chaine = replaceInChain(chainePrincipale, "\"", "");  chaine = replaceInChain(chaine, "\r\n", "");  chaine = replaceInChain(chaine, " ", " ");  return chaine;  }  private static String replaceInChain(String chainePrincipale, String patternARemplacer, String elementAInserer) {  return Pattern.compile(patternARemplacer).matcher(chainePrincipale).replaceAll(elementAInserer);  }  } | pack comparators |
|  |  |
| Lecteur de fichiers  lire le contenu d'un ou plusieurs fichiers et analyse ligne par ligne |  |
| /\*\*  \* LecteurFichiers <br>  \* <p>  \* Charge le contenu d'un ou plusieurs fichier et recherche/extrait de son  \* contenu la valeur spécifiée  \* </p>  \*  \* @author a-ramade  \* @since 2019  \*  \*/  public class LecteurFichiers {  public static final String PATH\_FICHIER = "C:\\Users\\a-ramade\\Desktop\\logs";  public static final String[] NOMS\_FICHIERS = { "gestion-des-envois-serveur-all.log" };  public static Pattern PATTERN = null;  public static List<String> linesTable = new ArrayList<>();  public static Set<String> linesSet = new HashSet<>();  /\*\*  \* lireEtExtraireLignes <br>  \* <p>  \* Charge le contenu d'un ou plusieurs fichier et recherche/extrait de son  \* contenu la valeur spécifiée  \* </p>  \*  \* @param valeurARetrouver  \* valeur à chercher dans le fichier  \* @param patternAExtraire  \* pattern à extraire si on ne veut pas afficher la ligne complète  \* (facultatif)  \*/  public static void lireEtExtraireLignes(String valeurARetrouver, String patternAExtraire) {  boolean extrairePattern = false;  if (Objects.nonNull(patternAExtraire) && !patternAExtraire.isEmpty()) {  PATTERN = Pattern.compile(patternAExtraire);  extrairePattern = true;  }  for (String nomFichier : NOMS\_FICHIERS) {  try (Stream<String> sourceLines = Files  .lines(Paths.get(String.format("%s\\%s", PATH\_FICHIER, nomFichier)))) {  List<String> lines = sourceLines.filter(line -> line.contains(valeurARetrouver))  .collect(Collectors.toList());  if (extrairePattern) {  linesSet.addAll(lines.stream().map(line -> CommonUtils.extraireSousChaine(PATTERN, line))  .collect(Collectors.toSet()));  } else {  linesSet.addAll(lines);  }  } catch (IOException e) {  e.printStackTrace();  }  }  afficherNbLines();  afficherLines();  }  public static void afficherNbLines() {  System.out.println(String.format("Nombre de lignes trouvées = %d", linesSet.size()));  }  public static void afficherLines() {  linesSet.stream().forEach(System.out::println);  }  } | pack filesreaders |
|  |  |
| Conversion  convertit un objet Java affiché dans les logs en objet JSON |  |
| public class JavaJson {  public static final String classeJava = "InfosEditiqueParam(risques=[CleMCParam(cle=11559\_\_C2019-1627\_\_idR1, codeDocument=ATT\_CARTE\_VERTE, dateConsultation=null), CleMCParam(cle=11559\_\_C2020-222\_\_idR12, codeDocument=ATT\_DAB\_PROPRIETAIRE, dateConsultation=2022-01-21), CleMCParam(cle=11559\_\_C2020-222\_\_idR13, codeDocument=ATT\_DAB\_PROPRIETAIRE\_NO, dateConsultation=null), CleMCParam(cle=11559\_\_C2020-222\_\_idR14, codeDocument=ATT\_DAB\_RISQUES\_LOCATIFS, dateConsultation=null)], contrats=null, numeroPage=1, nombreElementsPage=50)";  public static final List<String> TYPES\_NATIFS = Arrays.asList("true", "false", "null");  public static final String parentheseO = "\\(";  public static final String parentheseF = "\\)";  public static final String egal = "=";  public static final String virguleEspace = ", ";  public static final String accoladeO = "{";  public static final String accoladeF = "}";  public static final String deuxpoints = ":";  public static final String virgule = ",";  public static final String nomClasseRegex = "\\w\*\\{";  // public static final String texteRegex = "[{:\\s]\\p{javaLetter}+[:,}]";  public static final String dateTimeRegex = "\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}.\\d+";  public static final String texteRegex = "[\\p{javaLetter}\\w\\s\*'-]+";  public static void convertJavaToJson() {  System.out.println(classeJava);  // remplace les caractères de structures Java par des caractères JSON (= -> :, (  // -> { )  String result = Pattern.compile(parentheseO).matcher(classeJava).replaceAll(accoladeO);  result = Pattern.compile(parentheseF).matcher(result).replaceAll(accoladeF);  result = Pattern.compile(egal).matcher(result).replaceAll(deuxpoints);  result = Pattern.compile(virguleEspace).matcher(result).replaceAll(virgule);  // supprime le nom des classes  result = Pattern.compile(nomClasseRegex).matcher(result).replaceAll("\\{");  // remplace les dateTime par une simple date  result = Pattern.compile(dateTimeRegex).matcher(result).replaceAll("2019-01-01T00");  // ajout des " autour des valeurs de type String  Matcher m = Pattern.compile(texteRegex).matcher(result);  StringBuffer sb = new StringBuffer();  while (m.find()) {  if (!TYPES\_NATIFS.contains(m.group()) && !isNumeric(m.group())) {  m.appendReplacement(sb, "\"" + m.group() + "\"");  }  }  sb = m.appendTail(sb);  // affichage du résultat  System.out.println(sb.toString());  }  public static boolean isNumeric(String strNum) {  return strNum.matches("-?\\d+(\\.\\d+)?") && !strNum.startsWith("0");  }  } | pack converters |
|  |  |
| Gestion de fichiers  Déplacer, copier, renommer les fichiers dans un dossier |  |
| /\*\*  \* FilesAndFoldersManager<br>  \* <p>  \* Classe permettant de modifier les noms et les emplacements de fichiers et de  \* dossiers  \* </p>  \*  \* @author a-ramade  \* @since 09/2022  \*  \*/  public class FilesAndFoldersManager {  /\*\* Adresse du dossier principal \*/  private static String MAIN\_FOLDER\_PATH = "C:\\Users\\a-ramade\\Desktop\\TEST";  /\*\*  \* Type<br>  \* <p>  \* Défini le type d'élément à modifier  \* </p>  \* <div>  \* <ul>  \* <li>FOLDER</li>  \* <li>FILE</li>  \* </ul>  \* </div>  \*  \* @author a-ramade  \*  \*/  public enum Type {  FOLDER, FILE;  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* CONSTRUCTEURS \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* Constructeur par défaut  \*/  public FilesAndFoldersManager() {  super();  }  /\*\*  \* Constructeur avec adresse du dossier principal  \*  \* @param mainFolderPath  \* adresse du dossier principal  \*/  public FilesAndFoldersManager(String mainFolderPath) {  super();  MAIN\_FOLDER\_PATH = mainFolderPath;  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* FONCTIONS UTILITAIRES \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* isEmptyOrBlank<br>  \* <p>  \* Prédicat utilitaire renvoyant true si la chaine est nulle ou vide ou  \* contenant uniquement des caractères blancs  \* </p>  \*/  private static Predicate<String> isEmptyOrBlank = (str) -> str == null || str.trim().isEmpty();  /\*\*  \* getAbsolutePath<br>  \* <p>  \* Concatène l'adresse du dossier principal, l'adresse relative et le nom du  \* fichier permettant de créer le Path complet de l'élément  \* </p>  \*/  private static BiFunction<String, String, Path> getAbsolutePath = (folderOrFileName, relativePath) -> {  StringBuilder absolutePath = new StringBuilder(MAIN\_FOLDER\_PATH);  if (!isEmptyOrBlank.test(relativePath)) {  absolutePath.append("\\").append(relativePath);  }  if (!isEmptyOrBlank.test(folderOrFileName)) {  absolutePath.append("\\").append(folderOrFileName);  }  return Paths.get(absolutePath.toString());  };  /\*\*  \* getExtension<br>  \* <p>  \* Retourne l'extention d'un fichier à partir de son nom  \* </p>  \*/  private static Function<String, String> getExtension = (pathOrFileName) -> pathOrFileName  .substring(pathOrFileName.lastIndexOf("."));  /\*\*  \* relativePathExists<br>  \* <p>  \* Teste la validité d'une adresse relative  \* </p>  \*/  public static BiPredicate<String, Type> relativePathExists = (relativePath, type) -> {  Path completePath = getAbsolutePath.apply(null, relativePath);  return Files.exists(completePath)  && (type == Type.FOLDER ? Files.isDirectory(completePath) : Files.isRegularFile(completePath));  };  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* METHODES \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  /\*\*  \* createFolder<br>  \* <p>  \* Créé un dossier  \* </p>  \* <div>  \* <ul>  \* <li>vérifie la validité de l'emplacement cible</li>  \* <li>vérifie que le dossier n'existe pas déjà à cet emplacement</li>  \* <li>créé le nouveau dossier</li>  \* </ul>  \* </div>  \*  \* @param folderName  \* nom du dossier à créer  \* @param relativePath  \* adresse relative  \* @return this  \*/  public FilesAndFoldersManager createFolder(String folderName, String relativePath) {  try {  testRelativesPaths(relativePath, null);  Path newFolderPath = getAbsolutePath.apply(folderName, relativePath);  if (Files.notExists(newFolderPath)) {  Files.createDirectory(newFolderPath);  }  } catch (IOException e) {  e.printStackTrace();  }  return this;  }  /\*\*  \* copyAllFilesToFolder<br>  \* <p>  \* Créé une copie de tous les fichiers d'un emplacement source dans un  \* emplacement cible  \* </p>  \*  \* @param relativePathSource  \* emplacement source relatif  \* @param relativePathTarget  \* emplacement cible relatif  \* @param activateReplaceExistingOption  \* détermine si l'option écraser les fichers existant doit être  \* activée  \* @return this  \*/  public FilesAndFoldersManager copyAllFilesToFolder(String relativePathSource, String relativePathTarget,  boolean activateReplaceExistingOption) {  try {  testRelativesPaths(relativePathSource, relativePathTarget);  try (Stream<Path> paths = Files.walk(getAbsolutePath.apply(null, relativePathSource), 1)) {  paths.filter(Files::isRegularFile).forEach(filePath -> {  try {  if (activateReplaceExistingOption) {  Files.copy(filePath,  getAbsolutePath.apply(filePath.getFileName().toString(), relativePathTarget),  StandardCopyOption.REPLACE\_EXISTING);  } else {  Files.copy(filePath,  getAbsolutePath.apply(filePath.getFileName().toString(), relativePathTarget));  }  } catch (IOException e) {  e.printStackTrace();  }  });  } catch (IOException e) {  e.printStackTrace();  }  } catch (IOException e) {  e.printStackTrace();  }  return this;  }  /\*\*  \* moveAllFilesToFolder<br>  \* <p>  \* Déplace tous les fichiers d'un emplacement source vers un emplacement cible  \* </p>  \*  \* @param relativePathSource  \* emplacement source relatif  \* @param relativePathTarget  \* emplacement cible relatif  \* @return this  \*/  public FilesAndFoldersManager moveAllFilesToFolder(String relativePathSource, String relativePathTarget) {  this.filesMoveTo((fileName) -> fileName, relativePathSource, relativePathTarget, Type.FILE);  return this;  }  /\*\*  \* addIncrementation<br>  \* <p>  \* Ajoute une incrémentation à tous les noms des fichiers ou des dossiers  \* </p>  \* <u>la valeur de l'itération est obligatoirement placé au début du nom de  \* l'élément</u>  \*  \* @param firstStepOfIncrement  \* première valeur de l'itération  \* @param suffixSeparator  \* séparateur  \* @param relativePathSource  \* emplacement source relatif  \* @return this  \*  \* @see {@link Increment}  \*/  public FilesAndFoldersManager addIncrementation(String firstStepOfIncrement, String suffixSeparator,  String relativePathSource, Type type) {  Increment incr = new Increment(firstStepOfIncrement);  this.filesMoveTo((fileName) -> incr.increment("", "", suffixSeparator, fileName, ""), relativePathSource,  relativePathSource, type);  return this;  }  /\*\*  \* renameAndIncrement<br>  \* <p>  \* Renome et ajoute une incrémentation à tous les noms des fichiers ou des  \* dossiers  \* </p>  \*  \* @param prefix  \* préfixe  \* @param prefixSeparator  \* séparateur entre le préfixe et la valeur de l'itération  \* @param firstStepOfIncrement  \* première valeur de l'itération  \* @param suffixSeparator  \* séparateur entre le sufixe et la valeur de l'itération  \* @param suffix  \* sufixe  \* @param relativePathSource  \* emplacement source relatif  \* @param type  \* {@link Type#FILE} ou {@link Type#FOLDER}  \* @return this  \*  \* @see {@link Increment}  \*/  public FilesAndFoldersManager renameAndIncrement(String prefix, String prefixSeparator, String firstStepOfIncrement,  String suffixSeparator, String suffix, String relativePathSource, Type type) {  Increment incr = new Increment(firstStepOfIncrement);  this.filesMoveTo(  (fileName) -> incr.increment(prefix, prefixSeparator, suffixSeparator, suffix,  Type.FILE == type ? getExtension.apply(fileName) : ""),  relativePathSource, relativePathSource, type);  return this;  }  /\*\*  \* replacePatternInName<br>  \* <p>  \* Remplace ou supprime un motif dans les noms des fichiers ou des dossiers  \* </p>  \*  \* @param patternToReplace  \* motif à remplacer  \* @param replacementValue  \* valeur de remplacement  \* @param relativePathSource  \* emplacement source relatif  \* @param type  \* {@link Type#FILE} ou {@link Type#FOLDER}  \* @return this  \*/  public FilesAndFoldersManager replacePatternInName(String patternToReplace, String replacementValue,  String relativePathSource, Type type) {  Pattern pattern = Pattern.compile(patternToReplace);  this.filesMoveTo((fileName) -> pattern.matcher(fileName).replaceAll(replacementValue), relativePathSource,  relativePathSource, type);  return this;  }  /\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* METHODES METIERS \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*\*\*\*\* \*/  private void testRelativesPaths(String relativePathSource, String relativePathTarget) throws IOException {  if (!relativePathExists.test(relativePathSource, Type.FOLDER)) {  throw new IOException(String.format("The path '%s\\%s' doesn't exist or isn't a folder.", MAIN\_FOLDER\_PATH,  relativePathSource));  }  // teste la cible si != de la source (= vrai déplacement)  if (Objects.nonNull(relativePathTarget) && !relativePathSource.equals(relativePathTarget)  && !relativePathExists.test(relativePathTarget, Type.FOLDER)) {  throw new IOException(String.format("The path '%s\\%s' doesn't exist or isn't a folder.", MAIN\_FOLDER\_PATH,  relativePathTarget));  }  }  /\*\*  \* filesMoveTo<br>  \* <p>  \* Méthode utilisant {@link Files#move(Path, Path, java.nio.file.CopyOption...)}  \* permettant de déplacer ou de renomer les fichier et les dossiers  \* </p>  \* <div>  \* <ul>  \* <li>teste la validité de l'emplacement source</li>  \* <li>teste la validité de l'emplacement cible si nécéssaire</li>  \* <li>récupère le path de tous les éléments dans cet emplacement  \* <li>  \* <li>filtre ces éléments en fonction du type à modifier</li>  \* <li>applique la modification (renomage ou déplacement)</li>  \* </ul>  \* </div>  \*  \* @param getNewFileName  \* fonction de traitement a appliquer sur le nom de l'élément  \* @param relativePathSource  \* emplacement source relatif  \* @param relativePathTarget  \* emplacement cible relatif  \* @param type  \* {@link Type#FILE} ou {@link Type#FOLDER}  \*/  private void filesMoveTo(Function<String, String> getNewFileName, String relativePathSource,  String relativePathTarget, Type type) {  try {  testRelativesPaths(relativePathSource, relativePathTarget);  Path folderPath = getAbsolutePath.apply(null, relativePathSource);  // maxDepth = 1 => ne traverse que les éléments dans le dossier source (sans  // récursivité)  try (Stream<Path> paths = Files.walk(folderPath, 1)) {  // filtre les path en fonction du type d'éléments à modifier  paths.filter(path -> Type.FOLDER == type ? (Files.isDirectory(path) && !folderPath.equals(path))  : Files.isRegularFile(path)).forEach(filePath -> {  try {  // applique la fonction de transformation du nom de l'élément  // et genère le nouveau path avec le nom transformé  Files.move(filePath, getAbsolutePath.apply(  getNewFileName.apply(filePath.getFileName().toString()), relativePathTarget));  } catch (IOException e) {  e.printStackTrace();  }  });  } catch (IOException e) {  e.printStackTrace();  }  } catch (IOException e) {  e.printStackTrace();  }  }  } | pack Gestion |
|  |  |
|  |  |

**Exemples pratique de code Java**

Exemple de solutions pour problèmes particuliers à adapter selon le cas concret

**Problème : comment éliminer les doublons d'une liste avec discrimination personnalisée sans modifier la classe equals de l'objet**

|  |  |
| --- | --- |
| **Solution 1 /2**  utiliser un objet "Wrapper"  le Wrapper encapsule l'objet de référence et les critères de tri sont définies dans méthode equals du Wrapper  Avantages : utilisation de la méthode distinct() classique dans le stream  Inconvénient : création d'un objet supplémentaire | étapes d'encapsulation/désencapsulation dans le stream | |
| *// avec un Wrapper spécifique*  class WrapperEmployee {  private Employee e;  public WrapperEmployee(Employee e) {  this.e = e;  }  public Employee unwrap() {  return this.e;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  WrapperEmployee that = (WrapperEmployee) o;  return Objects.equals(e.getId(), that.e.getId());  }  @Override  public int hashCode() {  return Objects.hash(e.getId());  }  } | *//Utilisation*  List<Employee> unique = employee.stream()  .map(WrapperEmployee::new)  .distinct()  .map(WrapperEmployee::unwrap)  .collect(Collectors.toList()); |
| *// avec un Wrapper générique*  class Wrapper<T, U> {  private T t;  private Function<T, U> equalityFunction;  public Wrapper(T t, Function<T, U> equalityFunction) {  this.t = t;  this.equalityFunction = equalityFunction;  }  public T unwrap() {  return this.t;  }  @Override  public boolean equals(Object o) {  if (this == o) return true;  if (o == null || getClass() != o.getClass()) return false;  @SuppressWarnings("unchecked")  Wrapper<T, U> that = (Wrapper<T, U>) o;  return Objects.equals(equalityFunction.apply(this.t), that.equalityFunction.apply(that.t));  }  @Override  public int hashCode() {  return Objects.hash(equalityFunction.apply(this.t));  }  } | *//Utilisation*  List<Employee> unique = employee.stream()  .map(e -> new Wrapper<>(e, Employee::getId))  .distinct()  .map(WrapperEmployee::unwrap)  .collect(Collectors.toList()); |
| **Solution 2 /2**  utiliser un Predicat  utilisation d'un HashSet dans le prédicat pour permettre la discrimination  Avantages : pas de création d'objet suplémentaire, seulement d'un prédicat générique  Inconvénient : non utilisation de distinct() dans le stream => moins lisible | ne fonctionne pour une seule valeur discriminante | |
| *// Predicat générique*  public static <T> Predicate<T> distinctBy(Function<? super T, ?> f) {  Set<Object> objects = new ConcurrentHashSet<>();  return t -> objects.add(f.apply(t));  } | *//Utilisation*  employees.stream().filter(distinctBy(e -> e.getId)); |
|  |  |

**Exemples complets de codes en Java**

Portions de code complexes et complètes pouvant être réutilisées tel quel pour gagner du temps

|  |  |
| --- | --- |
| Key unique  création d'un ID ou d'une clé unique | Classe métier |
| import java.time.LocalDateTime;  import java.time.format.DateTimeFormatter;  import java.math.BigInteger;  import java.security.SecureRandom;  private static final DateTimeFormatter FORMATTER = DateTimeFormatter.ofPattern("yyMMddHHmmssSSS");  public String createKey() {  LocalDateTime dateTime = LocalDateTime.now();  StringBuilder newKey = new StringBuilder();  newKey.append(dateTime.format(FORMATTER));  newKey.append(getRandomName());  return newKey.toString();  }  private String getRandomName() {  SecureRandom random = new SecureRandom();  return new BigInteger(60, random).toString(12);  } |  |
|  |  |
|  |  |
|  |  |
| Affichage d'une fenêtre  défini la taille de la fenêtre et sa position centrée à l'écran | Vue |
| protected void defineSizeEtPosition(Dimension dim) {  this.setSize(dim);  Dimension ecranDim = Toolkit.getDefaultToolkit().getScreenSize();  this.setLocation((ecranDim.width - dim.width) / 2, (ecranDim.height - dim.height) / 2);  } |  |
|  |  |
| Sérialisation  sauvegarde d'un objet dans un fichier externe (avec gestion des erreurs) | Classe métier |
| public class SaveObject {  private static final String CHEMIN\_FICHIERS = "c:/ZfichiersTmp/";  private Map<String, String> erreurs = new HashMap<String, String>();  private String resultat;  public String getResultat() {  return resultat;  }  public Map<String, String> getErreurs() {  return erreurs;  }  public void saveObjectInFile(ClasseObject monObjet) {  ObjectOutputStream oos = null;  try {  // localization du fichier cible ou création  oos = new ObjectOutputStream(  new BufferedOutputStream(new FileOutputStream(new File(CHEMIN\_FICHIERS, "monFichier.txt"))));  // Ecriture de l'objet dans le fichier  oos.writeObject(monObjet);  } catch (IOException e) {  // captée lors d'une erreur d'écriture  erreurs.put("Erreur écriture", e.getMessage());  } finally {  // On s'assurer de fermeture du flux  System.out.print("Début de Finally");  try {  if (oos != null)  oos.close();  } catch (IOException e) {  erreurs.put("Erreur fermeture", e.getMessage());  }  }  if (erreurs.isEmpty()) {  resultat = "Copie de l'objet dans fichier réussie";  } else {  resultat = "Erreur lors de la copie de l'objet dans le fichier";  }  }  } |  |
|  |  |
| Déserialisation  lecture d'un objet sérialisé dans un fichier externe (avec gestion des erreurs) | Classe métier |
| public class ReadObject {  private static final String CHEMIN\_FICHIERS = "c:/ZfichiersTmp/";  private Map<String, String> erreurs = new HashMap<String, String>();  private String resultat;  public String getResultat() {  return resultat;  }  public Map<String, String> getErreurs() {  return erreurs;  }  public ClassObject readObjectInFile() {  ObjectInputStream ois = null;  ClassObject monObject = null;  try {  // localization du fichier cible  ois = new ObjectInputStream(  new BufferedInputStream(new FileInputStream(new File(CHEMIN\_FICHIERS, "monFichier.txt"))));  // Lecture de l'objet dans le fichier  monObject = (ClassObject) ois.readObject();  } catch (FileNotFoundException e1) {  // captée si ois ne trouve pas le fichier indiqué  erreurs.put("Fichier non trouvable", e1.getMessage());  } catch (IOException e2) {  // captée lors d'une erreur de lecture  erreurs.put("Erreur de lecture", e2.getMessage());  } catch (ClassNotFoundException e3) {  // captée si classe de l'objet inadaptée  erreurs.put("Erreur de cast", e3.getMessage());  } finally {  // On s'assurer de fermeture du flux  try {  if (ois != null)  ois.close();  } catch (IOException e) {  erreurs.put("Erreur de fermeture", e.getMessage());  }  }  if (erreurs.isEmpty()) {  resultat = "Lecture de l'objet dans le fichier réussie";  } else {  resultat = "Erreur lors de la lecture de l'objet dans le fichier";  }  return monObject;  }  } |  |
|  |  |
| Drag 'n Drop  TransferHandler personnalisé à attribuer à une JComponent pour activer le Drag n' Drop  *ici : pour* ***JLabel*** *(voir plus bas pour JTree)* |  |
| public class MyTransferHandler extends TransferHandler {  //autorise le type de données reçues par un Drop  public boolean canImport(TransferHandler.TransferSupport info) {  if(!info.isDataFlavorSupported(DataFlavor.stringFlavor)) { //type String uniquement  return false;  }  return true;  }    //insertion des données reçues par le Drop dans le composant  public boolean importData(TransferHandler.TransferSupport support) {  //controle si les données reçues sont d'un type autorisé  if(!canImport(support)) {  return false;  }    //récupération de l'objet Transferable contenant les données sources  Transferable data = support.getTransferable();  String str = "";    try {  str = (String)data.getTransferData(DataFlavor.stringFlavor);  } catch (UnsupportedFlavorException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }    //récupération du composant cible via le TransferSupport (ici un JLabel)  JLabel lbl = (JLabel)support.getComponent();  //Afectation des infos  lbl.setText(str);    return true;  }    //si action à faire une fois les données exportées vers un autre composant  public void exportDone(JComponent c, Transferable t, int action) {  //si MOVE et non COPY, on doit effacer les données du composant source si elles ont été déplacées  if(action == MOVE) {  ((JLabel)c).setText("");  }  }    //création de l'objet de type Transferable permettant le transfert des données entre les composants  protected Transferable createTransferable(JComponent c) {  //retourne un nouvel objet implémentant Transferable et contenant les données à déplacer (ici du texte)  return new StringSelection(((JLabel)c).getText());  }    //détermine le comportement du composant vas-à-vis du Drag n Drop  public int getSourceActions(JComponent c) {  return COPY; //copie des données uniquement  }    } | ATTENTION : l'option dans la méthode getSourceActions( ) doit être en accord avec celle définie dans le MouseListener ajouté au composant :  tHandler.exportAsDrag(monComp, e, TransferHandler.COPY); |
| *ici : méthode* importData( ) *adaptée pour une* ***JTree*** |  |
| //insertion des données reçues par le Drop dans le composant  public boolean importData(TransferHandler.TransferSupport support) {  if(!canImport(support))  return false;    //On récupère la localisation du Drop via un objet approprié  JTree.DropLocation dl = (JTree.DropLocation)support.getDropLocation();    //Infos pour pouvoir créer un nouvel élément  TreePath path = dl.getPath();  int index = dl.getChildIndex();    //Puis on récupère les données  Transferable data = support.getTransferable();  String str = "";    try {  str = (String)data.getTransferData(DataFlavor.stringFlavor);  } catch (UnsupportedFlavorException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }    //On ajoute le noeud  DefaultMutableTreeNode nouveau = new DefaultMutableTreeNode(str);  //On déduit le noeud parent via le chemain  DefaultMutableTreeNode parent = (DefaultMutableTreeNode)path.getLastPathComponent();    //On insere le noeud au bon endroit  DefaultTreeModel model = (DefaultTreeModel)this.tree.getModel();  index = (index == -1) ? model.getChildCount(path.getLastPathComponent()) : index;  model.insertNodeInto(nouveau, parent, index);    return true;  } |  |
|  |  |
|  |  |
| Arborescence de fichiers  création d'un arbre à partir des lecteurs racines de l'ordinateur | Classe métier |
| public class Fenetre extends JFrame {  private JTree arbre;  private DefaultMutableTreeNode racine;  public Fenetre2() {  this.setTitle("Les arbres");  this.setSize(300, 300);  this.setLocationRelativeTo(null);  this.setDefaultCloseOperation(EXIT\_ON\_CLOSE);    listRoot();    this.setVisible(true);  }    private void listRoot() {  this.racine = new DefaultMutableTreeNode();    for(File file : File.listRoots()){  DefaultMutableTreeNode lecteur = new DefaultMutableTreeNode(file.getAbsolutePath());  try {  for(File nom : file.listFiles()) {  DefaultMutableTreeNode node = new DefaultMutableTreeNode(nom.getName() + "\\");  lecteur.add(this.listFile(nom, node));  }  } catch (NullPointerException e) {    }    this.racine.add(lecteur);  }  arbre = new JTree(this.racine);    this.getContentPane().add(new JScrollPane(arbre));  }    private DefaultMutableTreeNode listFile(File file, DefaultMutableTreeNode node) {  int count = 0;    if(file.isFile())  return new DefaultMutableTreeNode(file.getName());  else {  File[] list = file.listFiles();  if(list == null)  return new DefaultMutableTreeNode(file.getName());  else {  for(File nom : list) {  count++;  //max 5 enfants par noeud  if(count < 5) {  DefaultMutableTreeNode subNode;  if(nom.isDirectory()){  subNode = new DefaultMutableTreeNode(nom.getName() + "\\");  node.add(this.listFile(nom, subNode));  } else {  subNode = new DefaultMutableTreeNode(nom.getName());  }  node.add(subNode);  }  }  return node;  }  }  } |  |
|  |  |
|  |  |
|  |  |