Programmation orientée objet
TP5 : rappels, paquets, flux, tests
(et c'est le dernier)

L3 MIAGE

26 octobre 2023

1 Quelques rappels

Note : les exercices commencent à la section 2. Nous vous recommandons chaudement de lire cette section avant de commencer les exercices.

Éditeurs de texte et IDE

Vous êtes libre d'utiliser l'éditeur de texte ou l'environnement de développement intégré (IDE) de votre choix. Parmi tous les choix disponibles, nous retenons les trois suivants

- Visual studio code
- Eclipse
- IntelliJ

Ces trois outils sont disponibles sur les ordinateurs de l'Université Paris-Nanterre au deuxième étage.

Recommandation : utiliser de préférence un éditeur de texte plutôt qu'un IDE. Notons que les IDE sont lourds puisqu'ils contiennent, entre autres, de nombreux outils pour vous faciliter l'écriture de programme. Notamment, ils permettent de générer du code à votre place (méthodes, constructeurs, getters, setters, etc.). Ceci est très pratique pour un informaticien expérimenté mais ne facilite pas l'apprentissage de la programmation. Il est important que vous sachiez reproduire le code, sans aide, qu'une machine est capable de générer automatiquement. Vous aurez ensuite toute votre carrière pour utiliser ces puissants outils.

Visual studio code : si vous choisissez d'utiliser cet éditeur de texte, vous pouvez installer *the Extension Pack for Java* qui contient des outils, entre autres, des outils de coloration syntaxique, un debugger et vous donne la possibilité d'exécuter un programme sans passer par un terminal.

Programme principal (compilation et exécution)

Voici une classe Main qui pourra vous servir tout au long de ce TP pour exécuter les programmes que vous avez réalisés. Vous êtes libre d'en modifier son contenu.

```
// Fichier Main.java
public class Main {
   public static void main(String[] args) {
      System.out.println("Voici un programme simple !");
   }
}
```

Pour compiler et exécuter votre programme en ligne de commande, vous pouvez utiliser les instructions suivantes dans le répertoire courant de votre fichier *Main.java*

```
javac Main.java
java Main
/* Resultat attendu:
Voici un programme simple !
```

Convention d'écriture (simple)

En java, il existe de nombreuses conventions d'écritures, mais nous vous demandons de respecter absolument les deux suivantes :

- suivre la syntaxe **camelCase** pour nommer les classes, interfaces, méthodes et variables. Si le nom est composé d'au moins deux mots, alors la première lettre du deuxième mot est une majuscule. Notons que la première lettre du nom d'une classe ou d'une interface doit toujours commencer par une majuscule tandis que la première lettre d'une méthode ou d'une variable doit toujours commencer par une minuscule.
- pour l'**indentation**, on utilise **4 espaces** (ou une tabulation équivalente à 4 espaces).

```
// fichier BienIndenter.java
class BienIndenter {
   // Le premier caractere de chaque champ commence par une
      minuscule. A chaque nouveau mot, le premier caractere
      commence par une majuscule. On evite d'utiliser le caractere
      '_' ou '-' pour separer les mots.
   int variable;
   String varLong;
   String varTresTresLong;

   // Le constructeur a toujours le meme nom que la classe
   BienIndenter(int v, String vL, String vTTL) {
      this.variable = v;
      this.varLong = vL;
      this.varTresTresLong = vTTL;
   }

   int calculerQuelqueChose() {
      return variable * 42;
   }
}
```

Champs et méthodes de classe

En java, il est possible de définir des champs associés à une classe C et non à un objet du type de la classe C. Vous pouvez voir ces champs comme des données globales qui n'existent qu'en un exemplaire. De la même manière, nous pouvons définir des méthodes de classes qui peuvent être appelées sans utiliser un objet de la classe. Voici quelques exemples :

```
class Main {
   // champs de classe ou champs statiques
   static in a = 55;
   static int b = 33;
   static int c = 44;

   // methode de classe ou methode statique
   static int somme(int x, int y) {
      return x + y;
   }

   // methode de classe ou methode statique
   static int produit(int x, int y) {
      return x * y;
   }

   // notons que la methode main est TOUJOURS une methode de classe
   public static void main(String[] args) {
      // Afin d'appeler une methode de classe f appartenant a la
         classe C, il faut utiliser l'instruction est: C.f(...)
      Main.somme(4181,999);

      // Afin d'acceder a un champ de classe x appartenant a la
         classe C, il faut utiliser l'instruction C.x
      Main.produit(Main.a, Main.b);
   }
}
```

Sortie

Voici quelques instructions contenant des méthodes d'affichage et des paramètres qui pourront vous être utile :

```
System.out.print("toto"); // Affiche le message toto sans retour a
   la ligne.

System.out.println("toto"); // Affiche le message toto avec un retour
a la ligne.

int rep = 42;
System.out.printf("La reponse est %d", rep); // La methode printf
   est similaire a la fonction printf en C. L'execution des
   intructions precedentes affichera le message "la reponse est 42".

System.out.print("1\n2\n3\n4"); // '\n' est un retour chariot c-a-d
   qu'il permet de retourner a la ligne. Ainsi, chaque chiffre sera
   affiche sur une ligne differente.
```

Les tableaux

Voici les instructions pour initialiser un tableau d'entiers et initialiser son contenu avec une boucle *for* :

```
int[] tab = new int[10];
for (int i = 0; i < 10; i++) {
   tab[i] = i + 1;
}
// Le tableau contient les valeurs 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Pour accéder à la taille d'un tableau, il existe l'instruction suivante ***tab.length***.

Chaîne de caractères

En java, vous pouvez représenter une chaîne de caractères par un tableau de caractères mais aussi en utilisant la classe String. Voici un exemple :

```
char[] msgTab = {'t', 'o', 't', 'o'};
String msgString = "toto";

// - pour un unique caractere on utilise les appostrophes ''
// - pour une sequence de caracteres (chaine de caracteres), on
   utilise les guillemets ""
```

Manipuler les entrées

En java, il est possible de récupérer les entrées saisies au clavier par l'intermédiaire de la classe *Scanner*. Voici quelques exemples :

```
import java.util.Scanner;

class Main {
   public static void main(String[] args) {
      Scanner entree = new Scanner(System.in);

      System.out.println("Saisir une chaine de caractere:");
      String s = entree.nextLine();
      System.out.println("Saisir un entier:");
      int a = entree.nextInt();
   }
}
```

Concaténation de chaîne de caractères

Pour la concaténation de chaîne de caractères en java, il est possible d'utiliser l'opérateur $+$. Voici quelques exemples :

```
System.out.println("Voici " + "mon" + " message !");

String s1 = "Ph'nglui mglw'nafh Cthulhu";
String s2 = "R'lyeh wgah'nagl fhtagn";
String s3 = s1 + " " + s2;
System.out.println(s3);
// Affichage attendu: Ph'nglui mglw'nafh Cthulhu R'lyeh wgah'nagl
   fhtagn
```

Si vous avez besoin de faire de nombreuses concaténations successives, par exemple au sein d'une boucle (*for*, *while*, ou autre), nous vous conseillons d'utiliser la classe *StringBuilder* qui est plus efficiente. Voici un exemple :

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i < 10; i++) {
   sb.append(i);
}
System.out.println(sb);
// Affichage attendu: 123456789
```

Exception

Définir une exception :

```
class MonException extends Exception {

   // Le constructeur par defaut
   MonException() {
      super();
   }

   // Constructeur pour creer une exception avec un message
      personnalise
   MonException(String msg) {
      super(msg);
   }

   // D'autres constructeurs possibles, exemple:
   MonException(int id, String nom) {
      super("L'utilisateur " + nom " avec l'identifiant " + id +
          " n'est pas enregistre dans la BDD");
   }
}
```

Lancer une exception :

```
class Jukebox {
   /* ... */
   void lireMorceau(String titre) {
      if (rechercheMorceau(titre) == false) {
         throw new JukeboxException(titre); // L'exception
            JukeboxException est levee. Si celle-ci n'est pas
            attrapee, alors le programme s'arretera.
      }
   }
   /* ... */
}
```

Attraper une exception :

```
class Main {
   /* ... */
   public static void main(String[] args) {
      Jukebox j = new Jukebox();
      try {
         j.lireMorceau("Ce titre n'existe pas !");
```

```
            System.out.println("Ce message ne s'affichera que si
                l'exception JukeboxException n'est pas levee !");
        } catch (JukeboxException e) {
            System.out.println("Les instructions de ce bloc ne seront
                executees que si l'exception JukeboxException est levee
                !");
            e.printStackTrace();
            System.out.println(e.getMessage());
        } finally {
            // Le bloc finally est optionnel
            System.out.print("Les instructions de ce bloc seront toutes
                executees apres les instructions du try ou du catch.");
        }
    }
    /* ... */
}
```

Lire dans un fichier

En java, il existe de nombreux outils pour lire des fichiers. Une des solutions les plus simples consiste à utiliser la classe *BufferedReader* qui fait partie de la bibliothèque de base de java. Voici un lien vers sa documentation : docs.oracle.com.

Voici un exemple d'ouverture d'un tampon pour la lecture d'un fichier nommé *readme.txt*.

```
// Attention: il faut attraper l'exception !
BufferedReader fichierTampon = new BufferedReader(new
    FileReader("readme.txt"));
```

Attention : le constructeur de *BufferedReader* prend en paramètre un objet *FileReader* qui peut lever une exception de type *FileNotFoundException* si le fichier n'est pas présent dans le répertoire courant.

Une fois le tampon pour la lecture du fichier initialisé, vous pouvez lire celui-ci ligne à ligne avec la méthode *readLine*. Cette méthode retourne un objet *String* si la fin du fichier n'a pas été atteinte, et *null* sinon.

```
// Attention: il faut attraper l'exception !
String test = fichierTampon.readLine();
```

Attention : la méthode *readLine* peut lever une exception de type *IOException*. Une telle exception peut être levée si, par exemple, le tampon a déjà été fermé.

Afin d'éviter la corruption d'un fichier ou la fuite de mémoire, il est préférable de fermer le tampon quand la lecture du fichier est terminée. Pour cela, on utilise la méthode *close*.

```
// Attention: il faut attraper l'exception !
fichierTampon.close();
```

Attention : la méthode *close* peut lever une exception de type *IOException*. Il faut aussi vérifier que le fichier tampon n'est pas *null*. Cela peut être le cas si celui-ci est présent dans le bloc d'instruction *finally* après un *try-catch* de la création du *BufferedReader*.

2 Les paquets

Java permet de regrouper des classes dans une collection (différent des collections Java) qui est appelé *paquet* (*package*). Les paquets sont pratiques pour organiser son travail et pour séparer son code de celui provenant de librairies externes.
Les paquets sont utiles pour différencier des classes ayants le même nom. Deux classes Date provenant de différents paquets seront considérées comme étant différentes :

```
java.sql.Date != java.util.Date
```

2.1 Importer une classe

Une classe peut utiliser toutes les classes provenant du même paquet sans avoir à les importer. C'est ce que vous faites naturellement quand tous vos fichiers Java sont placés dans le même dossier.
Il est aussi possible d'accéder à des classes publiques provenant d'un autre paquet de deux façons. La première consiste à donner le nom complet de la classe, c-à-d en ajoutant son paquet. Voici un exemple avec l'utilisation de l'interface List et d'une implémentation ArrayList :

```
java.util.List l = new java.util.ArrayList(10);
```

La seconde option plus courante et plus simple consiste à importer une classe avec import. Une fois que l'on a importé une classe, il n'est plus nécessaire de spécifier son paquet. Il est possible soit d'importer une classe spécifique, soit d'importer toutes les classes d'un paquet :

```
import java.util.*; // importe toutes les classes du paquet
    java.util comme List, ArrayList, Set.

import java.util.List; // importe uniquement la classe List
import java.util.Set; // importe uniquement la classe Set
```

Note : l'importation de toutes les classes d'un paquet n'a aucune incidence négative sur la taille de l'exécutable. La raison pour cela est que c'est le rôle du compilateur de localiser les classes dans les paquets. Ainsi, il remplace chaque classe par son nom complet avec son paquet.

```
List maListe = new ArrayList(10);
// L'instruction precedente sera remplacee par le compilateur par
    celle-ci:
java.util.List maListe = new java.util.ArrayList(10);
```

Cependant, l'utilisation de import .* peut poser des problèmes de collisions qui engendrent des erreurs à la compilation. Par exemple, le code ci-dessous ne compilera pas :

```
import java.sql.*
import java.util.*

/* ... */
Date d = new Date(); /* Erreur a la compilation: il existe une
    classe Date dans le paquet sql et dans le paquet util. Est-ce
    que le compilateur doit remplacer par
- java.util.sql (ou)
```

```
- java.util.Date */
/* ... */
```

Il est possible de résoudre cette collision en ajoutant l'import de la classe `Date` que vous souhaitez utiliser :

```
import java.sql.*;
import java.util.*;
import java.util.Date;
// la classe Date de sql ne sera pas importe
```

Et si on a besoin des deux classes, alors utiliser le nom de la classe avec son paquet :

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

Conseil : avec *Visual Studio Code*, il est possible de transformer un `import java.util.*` afin d'importer seulement les classes que vous avez utilisées. Pour cela, cliquer sur la petite ampoule jaune et sélectionner *organize import*. Note : l'accès à l'ampoule jaune se fait avec le raccourci « `ctrl + ;` ».

2.2 Imports statiques

Il est possible d'importer des variables et des méthodes statiques. Par exemple, la classe `Math` est souvent utilisée pour calculer des puissances, la racine carrée, ou pour obtenir la valeur de π. Son utilisation se fait comme suit :

```
import java.lang.Math;

Math.sqrt(11815); // calcul de la racine carre de 11815
Math.pow(2,18); // calcul de 2 a la puissance 18
Math.PI // valeur de pi
```

Pour apporter de la clarté au code, il est possible d'importer directement les méthodes et les variables statiques de la classe `Math` comme suit :

```
import static java.lang.Math.*; // Il est possible de remplacer *
    par la methode/variable a importer

sqrt(11815); // calcul de la racine carre de 11815
pow(2,18); // calcul de 2 a la puissance 18
PI // valeur de pi
```

2.3 Utiliser les paquets

Avant de montrer comment ajouter une classe à un paquet, nous proposons d'organiser les dossiers de ce TP comme suit :

```
TP5 (racine de votre dossier)
|-- src/
   |-- Main.java
|-- lib/
```

Le dossier `src` contient tous les fichiers `Java` de ce TP. Tous les fichiers à la racine de `src` appartiennent au paquet sans nom. Le seul fichier qui sera sans paquet sera votre fichier `Main.java`. Le dossier `lib` va contenir des librairies externes que nous introduirons plus tard.

Exercice 1

Créer la classe `Calculatrice` dans le paquet `tp5.util`. Voici la hiérarchie des dossiers que vous obtiendrez :

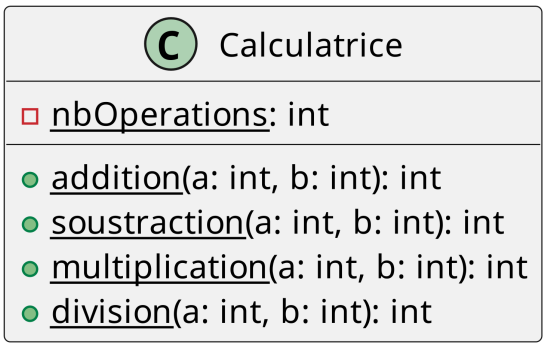
```
TP5 (racine de votre dossier)
|-- src/
   |-- tp5
      |-- util
         |-- Calculatrice.java
   |-- Main.java
|-- lib/
```

Le code de votre classe `Calculatrice` doit fonctionner comme suit :

```
package tp5.util

public class Calculatrice {
   /* ... */
}
```

Définir la classe `Calculatrice` afin de respecter le diagramme ci-dessous. Note : les méthodes et variables soulignées sont des méthodes statiques. La variable `nbOperations` sert à décompter le nombre d'opérations effectuées sur la calculatrice.



Vérification : pour tester votre classe `Calculatrice`, vous pouvez l'importer dans votre classe `Main` comme ci-dessous. N.B. pour compiler/exécuter votre programme `Main` en ligne de commande, il faudra se placer à la racine des sources du projet, c.-à-d. dans le répertoire `src`.

```
import tp5.util.Calculatrice;
```

Exercice 2

On souhaite que l'exception `CalculatriceException` soit levée lorsque la méthode division est appelée pour effectuer une division par zéro. Définir la classe `CalculatriceException` dans le paquet `tp5.exception`. Lever l'exception `CalculatriceException` lorsque cela est nécessaire dans la méthode `division`.

Vérification : Appeler la méthode `diviser` depuis la fonction main comme ceci :

```
Calculatrice.diviser(324245,5);
```

Si vous utilisez `Visual Studio Code`, alors vous pouvez générer un bloc de `try/catch` automatiquement. Pour cela, placer votre curseur sur la méthode `diviser`. Appuyer ensuite sur la petite ampoule jaune à gauche ou appuyer sur le raccourci « ctrl + ; ». Sélectionner « surround with try/catch ».
Si vous n'avez pas `Visual Studio Code`, alors il faudra le faire par vous-même (ou découvrir comment le faire avec votre outil) !

2.4 Visibilité dans les paquets

Précédemment, vous avez rencontré les mots clefs `private` et `public` pour modifier la visibilité des attributs/méthodes en dehors de la classe/objet. Il existe deux autres mots clefs de visibilité qui sont liés aux notions de paquets.

- `protected` : champ ou méthode accessible par toutes les sous-classes (héritage) et par toutes les classes du même paquet (mais pas par les sous-paquets).
- par défaut (quand aucun mot clef private/public/protected n'est spécifié) : champ ou méthode accessible par toutes les classes du même paquet (mais pas par les sous-paquets).

Pour jouer avec la visibilité des paquets, nous vous proposons l'exercice suivant :

Exercice 3

Objectif : Créer un système simple de gestion de bibliothèque en utilisant la visibilité par défaut (visible uniquement dans le paquet) pour gérer l'accès aux livres et aux utilisateurs.

Description :
Créer les deux classes ci-desous dans le même paquet nommé `tp5.bibliotheque`.

- `class Livre` : cette classe représente un livre et doit avoir les attributs tels que le numéro ISBN titre, l'auteur, l'année de publication, etc.
- `class Utilisateur` : cette classe représente un utilisateur de la bibliothèque, avec des attributs tels que le nom, le numéro d'adhérent, etc.

Dans le même paquet, créer une classe `Bibliotheque` qui servira de point d'entrée pour gérer la bibliothèque. La classe `Bibliotheque` doit avoir une liste de livres et une liste d'utilisateurs (vous pouvez utiliser des collections telles que des listes ou des tableaux pour stocker les livres et les utilisateurs). Définir des méthodes pour ajouter et retirer des livres de la bibliothèque, inscrire et désinscrire des utilisateurs, etc.

Créer une classe principale `GestionBiblio` à la racine de vos fichiers sources (dans le répertoire `src`). Les actions possibles devraient inclure l'ajout de livres à la bibliothèque, l'inscription d'utilisateurs, l'emprunt et le retour de livres, etc. Assurez-vous que la classe principale ne peut pas accéder directement aux attributs des livres ou des utilisateurs, car ils sont définis avec une visibilité par défaut.

Testez votre programme en utilisant la classe principale Main pour simuler des opérations de gestion de bibliothèque.
Voici un diagramme de classe résumant les classes que vous devez définir. N.B. le *symbole triangle* signifie que la méthode/attribut doit avoir une visibilité par défaut.

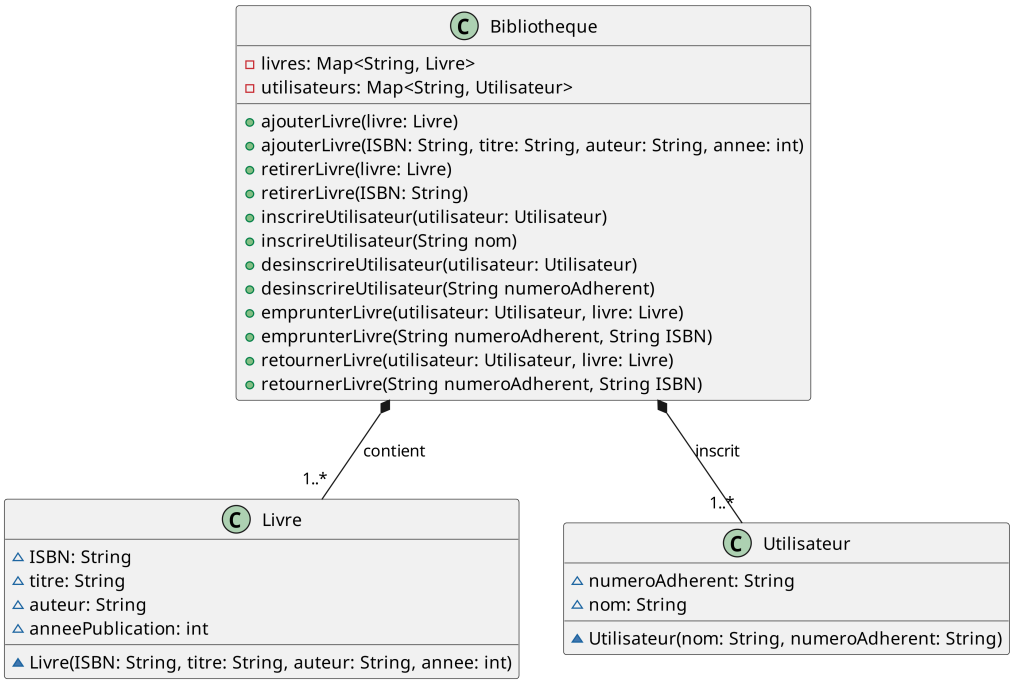


FIGURE 1 – Le diagramme de classes de l'application bibliothèque. Le triangle signifie que l'attribut ou la méthode a une visibilité par défaut.

3 Les flux (streams)

L'objectif de cette section est d'apprendre à utiliser les flux (`Streams`) de la bibliothèque `Java`. Les `Streams` sont disponibles depuis `Java8` qui est sorti en 2014. Ils permettent de traiter, manipuler et effectuer des opérations sur des séquences de données, notamment des collections (telles que des listes, des ensembles, des tableaux) et d'autres sources de données.

Les `Streams` peuvent servir à :

- **Filtrage et tri** : vous pouvez utiliser les `Streams` pour filtrer les éléments d'une collection en fonction de critères spécifiques, trier les éléments selon différents ordres, et effectuer des opérations de recherche avancées.
- **Réduction** : les `Streams` permettent d'effectuer des opérations de réduction (agrégation) telles que la sommation, le calcul de la moyenne, la recherche du minimum et du maximum, etc., sur les éléments d'une collection.
- **Parallélisme** : les `Streams` prennent en charge la programmation parallèle, ce qui signifie que vous pouvez traiter des données de manière concurrente, ce qui peut améliorer les performances pour les opérations intensives.
- **Lecture de données** : les `Streams` permettent de lire et de traiter des données en continu à partir de flux d'entrée, tels que des fichiers, des flux réseau, ou d'autres sources.
- **Lisibilité du code** : en utilisant des opérations de flux, vous pouvez écrire du code plus lisible, expressif pour manipuler les données, ce qui rend votre code

plus compréhensible et évite la nécessité d'écriture de boucles impératives complexes.

— **Économie de mémoire** : les `Streams` peuvent optimiser l'utilisation de la mémoire en évitant de stocker de grandes quantités de données intermédiaires en mémoire, car les données sont traitées de manière paresseuse (lazy) et au besoin.

En résumé, les `Streams` offrent un moyen efficace et déclaratif de manipuler et de traiter des données, ce qui facilite la programmation fonctionnelle et la rédaction de code plus propre et plus performant. Ils sont couramment utilisés pour effectuer des opérations de transformation, de filtration, de tri, de réduction, de lecture de données, et bien plus encore.

Lire des données et compter

À partir d'une collection de données, il est courant de parcourir ses éléments et de faire un traitement sur les données.
Exemple : nous disposons d'un fichier *lorem.txt* contenant du texte. Nous souhaitons compter le nombre de mots d'au moins 7 lettres dans le texte (le fichier est disponible sur *coursenligne*, vous pouvez le télécharger et le placer dans votre répertoire `src/`). Voici le code pour charger les mots du fichier dans une collection de type `List` :

```
String texte = Files.readStr(Paths.get("lorem.txt"));
List<String> listeMots = List.of(contents.split("\\PL+")); // Separe
    la chaine de caracteres en mots (la ponctuation est ignoree)
```

N.B nous avons utilisé la méthode `split` de la classe `String` qui permet de séparer une chaîne de caractères en une collection de chaîne de caractères en fonction du délimiteur choisi. Le délimiteur est spécifié à partir d'un *regex* qui est une expression régulière. Si cela vous intéresse, voici un lien vers un petit tutoriel sur les expressions régulières de `Java` : docs.oracle.com/javase/tutorial/essential/regex/char_classes.html. Voici un lien vers la liste exhaustive des expressions régulières disponibles : docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html.

Nous pouvons maintenant compter le nombre de mots que contient notre texte avec une boucle `foreach` :

```
cpt = 0;
for (String mot : listeMots) {
   if (mot.length() >= 7) {
      cpt++;
   }
}
```

Voici à quoi ressemble l'opération avec les `Streams` :

```
long cpt = listeMots.stream().filter(w -> w.length() > 12).count();
// Noter que w -> w.length() > 12 est une expression lambda que nous
    avons deja rencontre
```

Il est même possible de paralléliser cette opération en utilisant `parallelStream()` sur la collection :

```
long cpt = listeMots.parallelStream().filter(w -> w.length() >
    12).count();
// Les operations de filtres et de comptage vont etre parallelises
```

Comment fonctionne les Streams

Un flux semble similaire à une collection, vous permettant de transformer et de récupérer des données. Cependant, il existe des différences significatives :

- Un flux ne stocke pas ses éléments. Ils peuvent être stockés dans une collection sous-jacente ou générés à la demande.
- Les opérations sur un flux ne modifient pas leur source. Par exemple, la méthode de filtrage (`filter()`) ne supprime pas les éléments d'un flux, mais crée un nouveau flux dans lequel ils ne sont pas présents.
- Les opérations sur un flux sont paresseuses lorsque c'est possible. Cela signifie qu'elles ne sont pas exécutées tant que leur résultat n'est pas nécessaire. Par exemple, si vous demandez seulement les cinq premiers mots longs au lieu de tous, la méthode de filtrage (`filter()`) arrêtera de filtrer après le cinquième élément.

Lorsqu'on travaille avec les `Streams`, nous mettons une chaîne d'opérations en trois étapes :

1. Créer un flux.
2. Spécifier des opérations intermédiaires pour transformer le flux initial en d'autres flux, éventuellement en plusieurs étapes.
3. Appliquer une opération terminale pour produire un résultat. Cette opération force l'exécution des opérations paresseuses qui la précèdent. Après cela, le flux ne peut plus être utilisé.

Dans l'exemple précédent pour compter les mots, le flux est créé avec `Stream` ou `ParallelStream` à partir d'une collection de type `List`. La méthode `filter` transforme le flux pour ne garder que les mots de 7 caractères ou plus, et la méthode `count` dénombre le nombre de mots du flux filtré.

N.B. il est possible de convenir n'importe quel `Collection` en un `Stream` avec la méthode `stream()` ou parallelStream().

Exercice 1 : charger les livres

Télécharger le fichier *liste_livres.csv* sur *coursenligne*. En utilisant les informations précédentes, ajouter une méthode `chargerLivres` pour enregistrer tous les livres de ce fichier dans la classe `Bibliotheque`.

```
boolean chargerLivres(String cheminFichier);
```

Exercice 2 : plus ancien et plus récent

Déclarer deux méthodes `plusAncien()` et `plusRecent()` qui retourne, respectivement, le livre le plus ancien et le plus récent contenu dans la classe `Bibliotheque`.

```
Livre plusVieux();
Livre plusRecent();
```

Exercice 3 : premier livre qui commence par

Déclarer une méthode `premierQuiCommencePar()` qui prend en paramètre une chaînée de caractères `debutTitre`, et qui retourne la référence du premier livre dont le titre commence par la chaîne de caractères passé en argument.

```
Livre premierQuiCommencePar(String debutTitre);
```

Exercice 4 : compter les livres d'un auteur

Déclarer une méthode `compterLivres()` qui prend en paramètre le nom d'un auteur et qui retourne le nombre de livres écrit par cet auteur.

```
Livre compterLivres(String auteur);
```

Exercice 5 : quels sont les livres de x ?

Déclarer une méthode `listerLivres()` qui prend en paramètre le nom d'un auteur et qui retourne le nom des livres de l'auteur présent dans la bibliothèque. Exemple : auteur recherché Boris Vian.

```
/* Resultat attendu: L'ecume des jours, Les fourmis, L'herbe rouge,
   L'arrache-coeur, ... */

String listerLivres(String auteur);
```

Exercice 6 : par ordre de publication

Déclarer une méthode `afficherOrdreDate()` qui affiche les informations des livres de la bibliothèque par ordre de publication.

```
void afficherOrdreDate();
```