Université de Montpellier / Faculté Des Sciences

Structures de données

2019-2020 - HMIN215

Itérateurs, Lambda expressions et Streams

Marianne Huchard

1 Navigation de données complexes

Dans ce cours nous nous intéressons au problème de la navigation des données complexes avec une interface et des structures uniformes (au sens où la navigation s'effectue de la même manière quelle que soit la donnée complexe sur laquelle on navigue) et qui soient également pratiques et efficaces :

— collections, maps (dictionnaires associatifs),
— objets composites,
— flux, fichiers.

Java propose deux stratégies principales pour cette navigation, les *itérateurs* et, plus récemment les *streams*. Le tableau 1 résume ces deux stratégies principales, qui sont ensuite détaillées dans les deux sections suivantes.

Table 1 – Stratégies de navigation des données complexes en Java

Itérateurs	Streams
données plutôt **finies**	données finies ou **infinies**
itération **externe**	itération **interne**
sous le contrôle du **programmeur**	sous le contrôle de l'**interprète**
avec stockage des éléments	**sans stockage** des éléments
avec accès aux éléments	**sans accès** aux éléments

2 Les itérateurs

Un itérateur est un objet permettant de parcourir les éléments d'une collection et plus généralement les éléments internes à un autre objet complexe (qui est un composite) sans exposer l'implémentation. Cette forme de navigation sur la structure interne d'un objet par un autre objet (l'itérateur) est un patron

de conception connu (patron *Iterator*) et présenté dans le GOF (*Gang Of Four* pour désigner les quatre compères auteurs du livre) [1].

Ce patron de conception permet à l'utilisateur de l'objet complexe (collection ou objet composite) de ne pas connaître les détails de l'implémentation lorsqu'il veut simplement accéder aux éléments internes comme s'ils étaient dans une liste. Un des avantages est que la structure interne de l'objet peut changer (ainsi que le fonctionnement interne de l'itérateur) sans que le programme utilisateur n'ait à changer. Gamma et al. parlent d'*itération polymorphe* pour indiquer que grâce aux itérateurs on dispose d'une interface uniforme pour traverser des agrégats (et donc notamment des collections).

L'objet complexe donne au programme utilisateur l'accès à un ou plusieurs itérateurs par l'intermédiaire d'une méthode. Un itérateur classique offre des méthodes permettant de savoir quel est le premier élément, l'élément courant, le prochain élément, s'il existe un prochain élément, ou pour détruire l'élément courant (méthode moins fréquente).

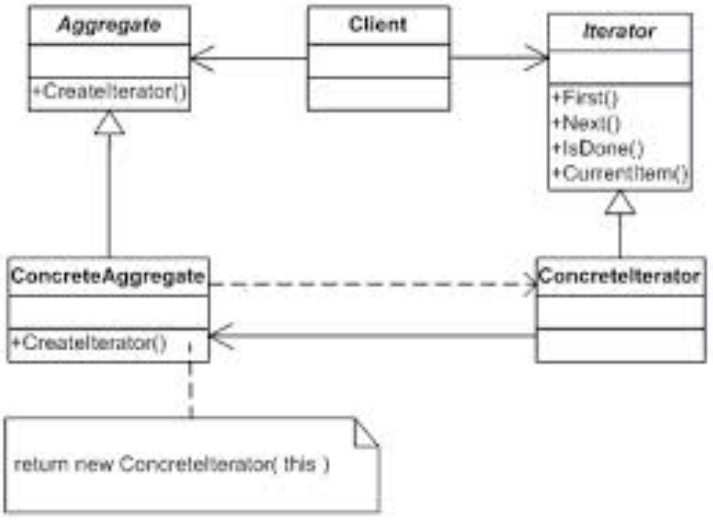

FIGURE 1 – **Patron Iterator (Gamma et al. 1994) : le programme client connaît l'agrégat et l'itérateur qui sert à le traverser. L'itérateur concret connaît ce qui lui est nécessaire de l'agrégat concret pour réussir à le traverser.**

3 Itérateurs en Java

3.1 L'interface `Iterator` de Java 1.8

On peut imaginer l'itérateur comme un curseur qui se déplace sur les parties internes de la collection ou de l'objet composite auxquelles on accède. L'interface propose trois méthodes qui permettent ce déplacement, ainsi qu'une méthode de retrait et une méthode permettant d'exécuter un traitement passé en paramètre sur la fin d'un itérateur. Elle définit donc le comportement des objets qui itèrent sur des objets complexes. L'interface est paramétrée par le type des objets retournés.

1. Design Patterns : Elements of Reusable Object-Oriented Software, By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Published Oct 31, 1994 by Addison-Wesley Professional

```
public interface Iterator<E>{
  boolean hasNext();  // Returns true if the iteration has more elements.
  E next();  // Returns the next element in the iteration.

  default void remove() {throw new  UnsupportedOperationException();}
  // Removes from the underlying collection the last element returned
                       // by the iterator (optional operation)

  default void forEachRemaining(Consumer<? super E> action)
      { while (hasNext())
         action.accept(next());} // Performs the given action for each remaining
         // element until all elements have been processed
         // or the action throws an exception.
}
```

3.2 L'interface `Iterable` (Java 1.8)

Cette interface définit le comportement des objets qui peuvent être la cible d'un itérateur, autrement dit sur lesquels on peut itérer. La méthode principale est `iterator()` qui retourne un itérateur sur l'objet complexe.

```
public interface Iterable<T>{
    Iterator<T>  iterator(); // Returns an iterator

    default void forEach(Consumer<? super T> action){
    // Performs the given action for each element of the Iterable until all
    // elements have been processed or the action throws an exception.
    for (T t : this)
         action.accept(t);
    }

    default Spliterator<T>  spliterator(){..}
    //Creates a Spliterator over the elements described by this Iterable.
}
```

3.3 Utilisation

Un premier exemple d'utilisation des itérateurs porte sur les collections de Java. Les collections que nous connaissons actuellement dérivent de l'interface `Collection`, qui dérive de l'interface `Iterable`. Dans une `ArrayList`, qui dérive de `Collection`, on héritera d'une implémentation de la méthode `iterator`. On y trouvera même une implémentation d'une forme spécialisée des itérateurs pour les listes (`ListIterator`).

On peut utiliser un itérateur de manière explicite, par exemple sur une liste d'étudiants `listeEtu` supposée non vide, contenant des objets d'une classe `Etudiant` disposant d'une méthode `moyenne()` :

```
List<Etudiant> listeEtu =  (...);
(...)
double moyenne=0;
```

```
   Iterator<Etudiant> ite = listeEtu.iterator();
   while (ite.hasNext())
      {  moyenne+=ite.next().moyenne();  }
   moyenne = moyenne/listeEtu.size(); // vérifier que la liste est non vide serait bienvenu,
                                      // c'est laissé à titre d'exercice
```

Il faut également savoir qu'il y a un itérateur caché (généré automatiquement) lorsque l'on appelle la boucle `"foreach"`. Cela signifie en particulier que dès que l'on définit un itérateur sur un objet complexe, on peut appeler sur celui-ci ce type d'itération :

```
   double moyenne=0;
   for (Etudiant e : listeEtu)
      { moyenne+=e.moyenne();  }
   moyenne = moyenne/listeEtu.size(); // vérifier que la liste est non vide serait bienvenu,
                                      // c'est laissé à titre d'exercice
```

On peut utiliser la méthode `remove` pendant l'itération. Par exemple, si on veut supprimer "Paolo" et "Jean", on peut écrire le code suivant.

```
Iterator<Etudiant> iter = listeEtu.iterator();
   while (iter.hasNext())
   {
      Etudiant e = iter.next();
      if (e.getNom().equals("Paolo")
               || e.getNom().equals("Jean") )
         iter.remove();
   }
```

Les listes disposent de `ListIterator`(s) qui offrent plus d'opérations pendant la navigation et les opérations `add`, `set` et `remove` seront rendues plus efficaces car elles s'appliquent à un endroit déterminé sur la liste.

```
public interface ListIterator<E> extends Iterator<E> {
   boolean hasNext();
   E next();
   boolean hasPrevious();
   E previous();
   int nextIndex();
   int previousIndex();
   void remove();
   void set(E o);
   void add(E o);
}
```

4 Définir son propre itérateur

Dans cette partie, nous montrons comment définir des itérateurs sur un objet complexe, qui est ici une représentation d'un distributeur de briques PEZ.

On se propose de définir un distributeur de bonbons qui s'inspire du système des distributeurs de briques PEZ (voir figure 2). Il s'agit d'un tube de 12 bonbons, que l'on récupère par le haut du tube grâce à un mécanisme qui fait remonter les bonbons au fur et à mesure.

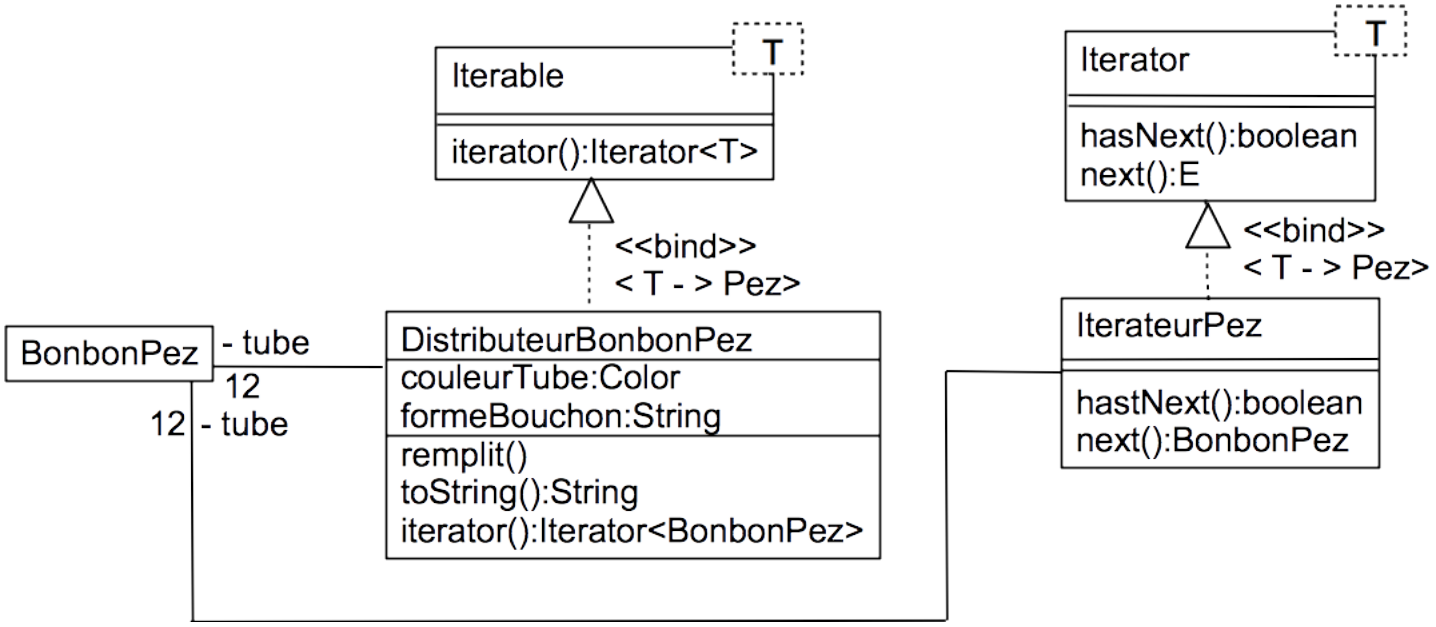

FIGURE 2 – **Patron Iterator appliqué au distributeur Pez**

Une classe minimale est donnée ci-dessous avec un programme `main` montrant le fonctionnement de l'itérateur. Le programme suppose l'existence d'une classe `BonbonPez` disposant d'un constructeur sans paramètre.

```
public class DistributeurBonbonPez implements Iterable<BonbonPez> {

	private Color couleurTube = Color.pink;
	private String formeBouchon = "Minnie";
	private BonbonPez[] tube = new BonbonPez[12];

	public void remplit()
	{
		for (int i = 0; i < 12; i++) tube[i] = new BonbonPez();
	}

	@Override
	public Iterator<BonbonPez> iterator() {
		return new IterateurPez(tube);
		// on donne parfois à l'itérateur la donnée sur laquelle il va travailler
```

```
        // (ici le tube)
        // ou parfois l'objet complet avec return new IterateurPez(this);
        // cela dépend du degré d'encapsulation mis en place
    }

    public String toString() {
        String s = "";
        for (BonbonPez b : tube) s+=b + " | ";
        return s;
    }
}
```

L'itérateur est un objet qui avance sur le tube (une sorte de pointeur généralisé, ou de curseur). Dans son implémentation, nous avons choisi d'effacer la case contenant la brique récupérée. C'est donc un itérateur "destructif". Si on désire qu'il ne le soit pas, on n'efface pas l'élément dans `next`. On peut le définir dans une classe à part mais on peut aussi en faire une classe interne ou une classe anonyme comme c'est souvent fait dans l'API Java.

```
public class IterateurPez implements Iterator<BonbonPez> {
    private BonbonPez[] tube; // la structure que l'on va traverser
    private int curseur = 0; // le curseur sur cette structure

    public IterateurPez(BonbonPez[] tube){this.tube = tube;}

    @Override
    public boolean hasNext() {
        return curseur < 12;
    }

    @Override
    public BonbonPez next() {
        BonbonPez bonbon = tube[curseur];
         tube[curseur]=null;
        curseur = curseur+1;
        return bonbon;
    }
}
```

Les deux classes une fois définies, on peut définir un programme simple montrant leur utilisation.

```
    public static void main(String[] argv)
    {
        DistributeurBonbonPez d= new DistributeurBonbonPez();
        d.remplit();
        System.out.println(d);

         for (BonbonPez b : d)
```

```
        // boucle for each utilisant l'itérateur et vidant le tube
        // cette écriture est possible car on a défini un itérateur
        {
                System.out.println(b);
                System.out.println("Apres retrait " + d);
        }
}
```

5 Streams et interfaces fonctionnelles

Les *streams*, ajoutés à partir de la version 1.8 de Java viennent compléter les collections et la manipulation de collections et de données plus généralement sur plusieurs aspects.

Rappelons que les collections et les itérateurs se préoccupent :

— du stockage des éléments,

— de l'accès aux éléments.

De leur côté, les *streams* :

— ne stockent pas leurs éléments, les éléments sont calculés à la demande, de manière paresseuse. On peut faire une analogie avec le visionnage de film en streaming à la demande, comparativement à disposer du film avec toutes les données chargées en mémoire,

— ne donnent pas un accès aux éléments,

— décrivent leur source de manière déclarative.

La notion de *stream* a été introduite principalement pour :

— traiter les données avec un langage plus déclaratif, à la manière de SQL,

— effectuer ce traitement de manière efficace, et sous certaines conditions, en parallèle, sans effort de programmation supplémentaire (le code sera optimisé).

Nous donnons une illustration simple de ces points qui seront ensuite explorés plus en détails.

Nous partons d'une classe représentant des *dossiers entreprises*, avec leur identification, année de création, adresse email et nombre de salariés et d'une classe *agence de coordination d'entreprises* contenant une liste de dossiers d'entreprises. Les classes peuvent être améliorées, notamment pour vérifier des invariants sur les attributs. Cela vous est laissé en exercice.

```
// classe représentant les dossiers entreprise
package streams_collectors_lambdas;

public class DossierEntreprise {
     private String identification;
     private int anneeCreation;
     private String emailAddress;
     private int nbSalaries;

     public DossierEntreprise(String identification, int anneeCreation,
                    String emailAddress, int nbSalaries) {
          this.identification = identification; this.anneeCreation = anneeCreation;
          this.emailAddress = emailAddress; this.nbSalaries = nbSalaries;
     }

  public String getIdentification() {return identification;}
  public void setIdentification(String identification)
                              {this.identification = identification;}
  public int getAnneeCreation() {return anneeCreation;}
  public void setAnneeCreation(int anneeCreation)
                              {this.anneeCreation = anneeCreation;}
  public String getEmailAddress() {return emailAddress;}
  public void setEmailAddress(String emailAddress)
                              {this.emailAddress = emailAddress;}
```

```
  public int getNbSalaries() {return nbSalaries;}
  public void setNbSalaries(int nbSalaries)
                            {this.nbSalaries = nbSalaries;}

  public boolean sup2012(){ return this.getAnneeCreation() >=2012; }
}

// classe représentant une agence de coordination d'entreprises
import java.util.ArrayList;
import java.util.OptionalDouble;
import java.util.stream.Collectors;

public class AgenceCoordination {
   private ArrayList<DossierEntreprise> listeEntreprises = new ArrayList<>();

   public AgenceCoordination(){}

   public void accueilleEntreprise(DossierEntreprise d){
      if (! listeEntreprises.contains(d))
         listeEntreprises.add(d);
   }
}
```

A présent, imaginons que nous voulions calculer le nombre moyen de salariés des entreprises créées depuis 2012. On peut utiliser de manière classique les opérations sur les listes (notamment l'itérateur) comme montré ci-dessous.

```
// Dans AgenceCoordination
     public double nbMoyenSalarie2012CO(){
          double sommeNbSalaries = 0;
          int nbEntreprise2012 = 0;
          for (DossierEntreprise d : listeEntreprises)
               if (d.getAnneeCreation()>=2012)
                    {
                         sommeNbSalaries += d.getNbSalaries();
                         nbEntreprise2012++;
                    }
           if (nbEntreprise2012 >0)
                   return sommeNbSalaries/nbEntreprise2012;
          else return 0;
     }
```

Une méthode basée sur les *streams* va consister à récupérer les données de la liste dans un *stream* (par la méthode `stream`), puis à lui appliquer des opérations de filtrage (filter), de projection (map), puis d'agrégation (average). La figure 3 schématise ce pipeline de traitements.

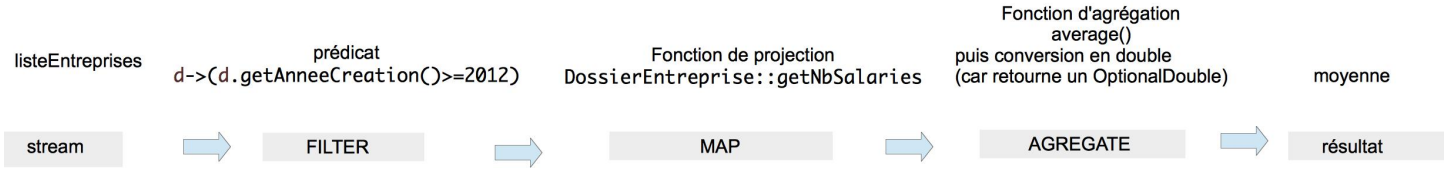

FIGURE 3 – Pipeline de traitement sur `listeEntreprises`

```
public double nbMoyenSalarie2012ST(){
     return listeEntreprises
          .stream()
          .filter(d->(d.getAnneeCreation()>=2012)) // lambda expression
          .mapToDouble(DossierEntreprise::getNbSalaries) // référence de méthode
          .average()
          .getAsDouble();
 }
```

Même si ce n'est **absolument pas réalisé ainsi en pratique** (voir plus loin), cela revient du point de vue logique, pour le programme suivant :

```
AgenceCoordination ag = new AgenceCoordination();
DossierEntreprise e1 = new DossierEntreprise("00034", 2012, "cheminvert@op.fr",10);
DossierEntreprise e2 =  new DossierEntreprise("00035", 2008, "caramelo@op.fr",10);
DossierEntreprise e3 =  new DossierEntreprise("00036", 2014, "pizzaDelice@op.fr",20);
ag.accueilleEntreprise(e1);
ag.accueilleEntreprise(e2);
ag.accueilleEntreprise(e3);
System.out.println(ag.nbMoyenSalarie2012ST());
```

à calculer successivement :

```
// entreprises creees apres 2012 - correspond au filtre
          e1   e3
// recuperation des nombres de salaries - correspond à la projection
          10   20
// somme et moyenne des nombres de salaries - correspond à l'agrégation
          15.0
```

Deux catégories principales d'opérations peuvent être utilisées :

— les opérations intermédiaires, qui produisent un *stream* à partir d'un autre *stream* (comme `filter, sort, map`) et peuvent être connectées ensemble,

— les opérations terminales (comme `collect`), qui terminent effectivement le calcul et produisent un résultat.

6 Efficacité

Deux éléments de mise en œuvre du calcul lui donnent son efficacité, le calcul paresseux et la parallélisation. C'est pour ces deux raisons que la vue logique donnée du calcul ci-dessus ne correspond pas à sa mise en œuvre pratique.

6.1 Calcul paresseux (Lazy)

Le calcul des opérations intermédiaires est effectué si possible en une seule passe (c'est l'interpréteur qui s'en charge), alors que dans la plupart des calculs sur les collections, si on décide de décomposer le calcul en plusieurs passes, il n'y a pas d'optimisation effectuée. Pour les pipelines de *streams*, le calcul est lancé lorsque l'opération terminale est atteinte.

Par exemple on aurait pu écrire de manière non efficace le calcul précédent comme suit, et il serait effectivement exécuté comme on l'a demandé, alors que ce n'est pas une bonne approche en pratique puisqu'elle va effectuer plusieurs parcours et créations successifs de listes, là où un seul suffit.

```
public double nbMoyenSalarie2012COv2(){
     if (listeEntreprises.isEmpty()) return 0;
     double sommeNbSalaries = 0;

     // entreprises creees apres 2012 - correspond au filtre
     ArrayList<DossierEntreprise> listeEnt2012 = new ArrayList<>();
     for (DossierEntreprise d : listeEntreprises)
          if (d.getAnneeCreation()>=2012)
               listeEnt2012.add(d);

     // recuperation des nombres de salaries - correspond à la projection
     ArrayList<Integer> listeDesNbSalaries = new ArrayList<>();
     for (DossierEntreprise d : listeEnt2012)
          listeDesNbSalaries.add(d.getNbSalaries());

     // somme des nombres de salaries - correspond à l'agrégation
     for (Integer d : listeDesNbSalaries)
          sommeNbSalaries += d;

if (listeDesNbSalaries.size() > 0)
     return sommeNbSalaries/listeDesNbSalaries.size();
else return 0;
}
```

6.2 Parallélisation

Pour paralléliser, il suffit de remplacer `stream()` par `parallelStream()`. L'API des *streams* s'occupera de décomposer la requête pour qu'elle soit exécutée en parallèle si l'architecture machine le permet.

```
public double nbMoyenSalarie2012ST(){
   return listeEntreprises
      .parallelStream()
      .filter(d->(d.getAnneeCreation()>=2012))
      .mapToDouble(DossierEntreprise::getNbSalaries)
      .average()
      .getAsDouble()
   ;
}
```

7 Lambda expressions, références de méthodes et de constructeurs, interfaces fonctionnelles

Java 1.8 a introduit divers procédés pour passer des opérations en paramètres ou les manipuler par l'intermédiaire de variables : les interfaces fonctionnelles, les lambda expressions et les références de méthodes et de constructeurs.

7.1 Interface fonctionnelle

Une interface fonctionnelle est une interface qui a exactement une méthode abstraite qui n'a pas la même signature qu'une méthode de la classe `Object` (elle peut avoir par ailleurs des méthodes qui ont les mêmes signatures que des méthodes de la classe `Object`). Une interface fonctionnelle peut en réalité hériter de plusieurs méthodes abstraites qui peuvent se substituer les unes aux autres, et à condition que l'une soit plus spécifique que les autres. Elle peut avoir d'autres méthodes des sortes `default` ou `static`.

Une interface fonctionnelle peut être introduite par l'annotation @FunctionalInterface (c'est recommandé mais ce n'est pas obligatoire).

```
@Retention(value=RUNTIME)
@Target(value=TYPE)
public @interface FunctionalInterface
```

Une instance d'une interface fonctionnelle peut être une lambda expression, une référence de méthode ou encore une référence de constructeur.

Ce sont ces instances que l'on peut passer en paramètre à des opérations comme `filter`, `map` ou `forEach` dans le cadre des streams, opérations qui attendent des opérations en paramètres.

Toute une hiérarchie d'interfaces fonctionnelles existe en Java et sert notamment à écrire les signatures de ces méthodes qui attendent des opérations en paramètres. Elles se trouvent dans le paquetage `java.util.function`.

En voici un exemple dont nous ne présentons que la méthode abstraite qui justifie que ce soit une interface fonctionnelle.

```
@FunctionalInterface
public interface Predicate<T>{
  boolean test(T t) // Evaluates this predicate on the given argument
}
```

Cette interface `Predicate` sert à décrire le paramètre de la méthode `filter` des streams. La signature vous est présentée ci-dessous.

```
Stream<T>  filter(Predicate<? super T> predicate)
Returns a stream consisting of the elements of this stream that match the given predicate.
```

On peut utiliser l'interface `Predicate` pour déclarer explicitement une variable contenant un prédicat.

```
public double nbMoyenSalarie2012STv3(){
   Predicate<DossierEntreprise> p = DossierEntreprise::sup2012;
  // ou Predicate<DossierEntreprise> p = d->(d.getAnneeCreation()>=2012);

  return listeEntreprises
     .stream()
     .filter(p)
     .collect(Collectors.averagingInt(DossierEntreprise::getNbSalaries)) ;
}
```

7.2 Lambda expressions

Dans l'exemple précédent, nous avons utilisé une lambda expression, `d->(d.getAnneeCreation()>=2012)`, qui représente une fonction qui associe à `d` le booléen `d.getAnneeCreation()>=2012`.

On peut comprendre une lambda expression comme une fonction anonyme, définie en Java avec la syntaxe suivante :

```
          (liste de parametres)    ->     body
```

Les lambda expressions ont de multiples usages en Java, en général pour passer des opérations en paramètres à des méthodes. Il y a un certain nombre d'exemples dans l'API. Par exemple, depuis Java 1.8 une manière simple de mettre en œuvre un tri sur une liste consiste à appeler la méthode `sort` avec une lambda expression en paramètre. Ci-dessous nous vous montrons comment trier la liste d'entreprises selon l'ordre alphabétique des adresses mail.

```
public void trie(){
    Collections.sort(this.listeEntreprises,
            (de1,de2)->de1.getEmailAddress().compareTo(de2.getEmailAddress()));
}
```

Voici ensuite une méthode pour afficher les adresses mails des entreprises qui peut s'écrire comme suit, grâce à la nouvelle méthode `forEach` de l'interface `List`.

```
public void afficheEmails(){
    this.listeEntreprises.forEach(e ->System.out.println(e.getEmailAddress()));
}
```

Voici d'autres exemples de lambda expressions, montrant les éléments qui peuvent être optionnels (les accolades et le `return`, les types des paramètres, les parenthèses autour de la liste de paramètres quand il n'y a qu'un paramètre) :

```
// avec type de paramètre
(DossierEntreprise d) -> d.getAnneeCreation()>=2012

// avec un body sous forme de bloc d'instructions et sans indiquer de type de paramètre
d -> { return d.getAnneeCreation()>=2012; }

// avec  deux paramètres sans indiquer leur type
(a,b) -> a+b

// avec les deux paramètres typés
(int a, int b) -> a+b
```

Les lambda expressions peuvent accéder à certains éléments de leur environnement englobant comme des méthodes, des variables locales (qui ne peuvent être modifiées) ou des attributs (qui peuvent être modifiés). Dans l'exemple suivant, la lambda expression accède à la variable locale année (mais ne pourrait pas la modifier). Ici on cherche à afficher les emails des entreprises créées après 2012 :

```
public void afficheMails(){
  int annee = 2012;
  listeEntreprises
      .stream()
      .filter(d -> d.getAnneeCreation()>=annee)
      .map(d -> d.getEmailAddress())
      .forEach(email -> System.out.println("annee >"+annee+" "+email));
}
```

Et dans le prochain exemple, la lambda expression accède à l'attribut année et peut le modifier, ce qui a pour effet ici d'afficher l'email de la première entreprise créée depuis 2012, puis uniquement les emails des autres entreprises créées depuis 2014 :

```
public int annee = 2012;

public void afficheMails(){
   listeEntreprises
      .stream()
      .filter(d -> d.getAnneeCreation()>=annee)
      .map(d -> d.getEmailAddress())
      .forEach(email -> {System.out.println("annee >"+annee+" "+email); annee = 2014;});
   }
```

L'affichage des emails des entreprises créées après 2012 peut cependant s'écrire de manière plus simple comme suit sans capture d'environnement.

```
public void afficheMails(){
   listeEntreprises
      .stream()
      .filter(d -> d.getAnneeCreation()>=2012)
      .map(d -> d.getEmailAddress())
      .forEach(email -> System.out.println(email));
}
```

7.3 Références de méthodes ou de constructeurs

Les expressions de référence de méthodes ou de constructeurs qui nous intéressent principalement ici sont formées à l'aide de l'opérateur :: (opérateur de résolution de portée). Dans les versions simples, à leur gauche se trouve un nom de type, et à leur droite un nom d'opération. Il y a des expressions plus complexes. Quelques exemples sont présentés ci-dessous.

```
System.out::println
List<String>::size
ArrayList<String>::new
```

8 Opérations sur les *streams*

De manière générale, voici ce que vous pourrez trouver comme types d'opérations sur les *streams* :

- opérations intermédiaires (créant d'autres *streams*)
  - le filtrage : par un prédicat `filter(predicate)`, en enlevant les doublons `distinct()`, en réduisant la taille `limit(n)`, en sautant les n premiers éléments `skip(n)`,
  - la projection, qui se base sur une fonction, `map(fonction)`, ou qui transforme en *streams* spécialisés `mapToInt(fonction)` ou `mapToDouble(fonction)`.
- opérations terminales
  - l'application d'une procédure (de type `void`) à tous les éléments du flot : `forEach(fonction)`,
  - les recherches et appariements (sous forme d'opération terminale) : par un prédicat, pour vérifier que tous les éléments le satisfont `allMatch(predicate)`, ou qu'un élément le satisfait `anyMatch(predicate)`, ou pour récupérer le premier élément qui le satisfait `findFirst(predicate)` ou pour récupérer n'importe quel élément qui le satisfait `findAny(predicate)`,
  - la réduction par `collect(Collector)`, où `Collector` est une opération de réduction ou de regroupement d'éléments de la collection (un exemple est donné plus loin),
  - la réduction qui applique une opération de manière répétitive : `reduce(valeur d'accumulation, fonction d'accumulation)`, et sur les *streams* numériques on dispose également d'opérations spécialisées comme `sum()` ou `average()`.

9 Créer des *streams*

Il existe plusieurs manières de créer des *streams* :
— à partir de valeurs données en extension
`Stream<Integer> pairs = Stream.of(2,4,6,8);`
ou placées dans un tableau
`int[] tabDePairs = {2,4,6,8};`
`IntStream pairs = Arrays.stream(pairs);`
— à partir d'une collection comme vu précédemment,
— à partir d'un fichier,
— en produisant des éléments à la demande à partir d'une fonction. Cette production peut être "infinie".
Les deux derniers cas sont présentés ci-dessous.

9.1 Création d'un `stream` depuis un fichier

Nous illustrons tout d'abord la création d'un *stream* à partir d'un fichier, par l'opération `lines`. On y voit également une utilisation de l'opération `collect` pour créer une autre collection, ici un dictionnaire associatif dont les clefs sont des années de parution et les valeurs des listes de couples (année de parution, titre de livre).

```
// contenu de fichier annee/titre, le fichier est stocké dans ./data/books
1830 / Le rouge et le noir
1837 | Le rose et le vert
1830 / le livre des oracles

// Code qui récupère les lignes du fichier (utilise la classe Paire qui suit)
// et les place dans une Map
Map<Object,List<Paire>> m=
   Files
   .lines( Paths.get("data", "books"))
   .map(   (String line) -> {
                  String[] elements = line.split("[|/]") ;
                  String annee = elements[0];
                  String titre = elements[1];
                  return new Paire(annee,titre);
                  }
            )
   .collect(Collectors.groupingBy(Paire::getPremier));
System.out.println(m);
// affiche la map créée
//{1830=[Paire [premier=1830, second=Le rouge et le noir],
//       Paire [premier=1830, second=le livre des oracles]],
//1837=[Paire [premier=1837, second=Le rose et le vert]]}
```

```
class Paire{
   public String premier, second;
   public Paire(String premier, String second) {
      this.premier = premier;
      this.second = second;
   }
   public String getPremier() {return premier;}
   public void setPremier(String premier) {this.premier = premier;}
   public String getSecond() {return second;}
   public void setSecond(String second) {this.second = second;}
   public String toString() {return "Paire [premier=" + premier + ",
                                                    second=" + second + "]";}
}
```

Pour l'analyse des fichiers, dont les éléments sont souvent récupérés sous forme de chaînes de caractères, il est recommandé d'utiliser la manipulation des expressions régulières permise par le paquetage `java.util.regex` ou par des méthodes de la classe `java.lang.String`. Ces manipulations consistent à vérifier si une chaîne de caractères s'apparie avec une certaine expression régulière (variantes de méthodes `matches`), permettent de la découper (variantes de méthodes `split`) ou de la modifier (variantes de méthodes `replace`). Ci-dessus, l'expression `"[|/]"` indique que l'on va découper chaque ligne suivant les caractères `|` (pipe) ou `/` (divise). La grammaire d'écriture des expressions régulières est très riche, nous vous conseillons de consulter la documentation lorsque vous avez de tels problèmes à traiter.

9.2 Création d'un `stream` à partir d'une fonction

Enfin nous montrons comment créer un *stream* à partir d'une fonction. La seconde instruction permet d'afficher les éléments du flot dans une limite de 4. Le flot étant ici infini, si on ne met pas d'appel à l'opération `limit(4)`, le programme ne s'arrête pas.

```
Stream<Integer> pairs = Stream.iterate(0, n -> n + 2);
pairs.limit(4).forEach(System.out::println);
```