le cnam

Sujet UE NFA035 : Programmation Java : bibliothèques et patterns

Correction commentée

Année universitaire 2015 – 2016
Examen 2e session : 6/9/2016
Responsable : Serge ROSMORDUC

Durée : 3 heures

Tout document *papier* autorisé. Tout support électronique est interdit : pas d'ordinateur, de tablette, de liseuse…

Les téléphones mobiles et autres équipements communicants doivent être éteints et rangés dans les sacs pendant toute la durée de l'épreuve.

Le barême est donné à titre indicatif ; il est susceptible de modifications.

Sujet de 7 pages, celle-ci comprise.

Exercice 1 *10 points (les collections)*

Cet exercice vise à voir si vous savez utiliser les collections dans un contexte de modélisation objet (avec plusieurs classes, donc). Une première étape, très importante, est de comprendre comment les différentes classes se combinent les unes aux autres.

Je vous suggère de faire au brouillon un petit schéma avec les classes, leurs variables d'instance et éventuellement la liste de leurs principales méthodes.

On souhaite gérer l'installation de logiciels sur une machine. Un logiciel est décrit par un nom, un numéro de version et par l'ensemble des logiciels dont il dépend : ils sont nécessaires à son installation/utilisation. Exemple : Netbeans a besoin de JDK pour être installé/utilisé. Donc, Netbeans dépend de JDK. A la création, l'ensemble de dépendances d'un logiciel est vide. Elles sont ajoutées une à une via la méthode `ajoutDependance`. Un objet de la classe `InstallMachine` contiendra tous les logiciels installés sur la machine. On se donne les classes suivantes :

```
public class Logiciel {
   private String nom;
   private int version;
   // INVARIANT: dependances ne contient pas de doublons.
   private Set<Logiciel> dependances = new HashSet<Logiciel>();

   public Logiciel(String n, int v) {
        this.nom=n; this.version=v;
   }
   public String getNom(){
        return nom;
   }
   public int getVersion(){
        return version;
   }
   public boolean dependDe(Logiciel p){
        return dependances.contains(p);
   }
   public boolean ajoutDependance(Logiciel l){
       boolean added = dependances.add(l);
       return added;
   }
   public Set<Logiciel> getDependances(){
        return new HashSet<Logiciel>(dependances);
  }
 }
```

*Un point intéressant dans la dernière méthode : en appelant `return new HashSet<>(dependances)` on retourne une **copie** de la collection interne `dependances`. L'intérêt de la chose est que l'utilisateur de la méthode peut s'il le souhaite modifier le contenu de cette copie sans modifier la collection d'origine.*

```
public class InstallMachine {
   // NOMS + LOGIELS INSTALLES SUR LA MACHINE
   private HashMap<String,Logiciel> installes= new HashMap<String,Logiciel>();

   /**
    * Dit si le logiciel nommé nl est installé.
```

```
     * @param nl un nom de logiciel
     * @return true si et seulement si
     *   une version de nl est installée.
     */
    public boolean estInstalle(String nl){
       return installes.containsKey(nl);
    }
    public boolean estInstalle(Logiciel l){
       ... // A COMPLETER
    }
    /**
     * Enregistre un logiciel comme installé,
     * si c'est possible.
     * un logiciel peut être installé s'il ne l'est pas déjà,
     * et si ses dépendances sont installées, avec les bonnes
     * versions.
     * post-condition: si le logiciel peut être
     * installé, il est enregistré dans la map.
     * @return true si le logiciel a pu être installé.
     */
    public boolean installer(Logiciel l){
       if (estInstalle(l)){
          return false;
    } ... // COMPLETER
}
```

Note importante : Sauf mention du contraire, on suppose dans toutes les questions que le code donné est complet et se comporte correctement, et en particulier, que les invariants sont respectés.

Note importante 2 : Supposons que Netbeans dépende de JDK et que JDK dépende de `gcc`. On dit que Netbeans dépend *directement* de JDK et *indirectement* de gcc. Dans tout cet exercice **on ne travaille que sur les dépendances directes** : celles figurant explicitement dans la liste de dépendances d'un logiciel.

Question 1.1 *(1 point)*

Dans la classe `InstallMachine` on vous demande de compléter le code de la méthode `estInstalle(Logiciel l)`. Cette méthode doit tester non seulement le nom du logiciel mais aussi son numéro de version. Par exemple, si JDK 6 est installé, un appel qui teste si JDK 8 est installé doit renvoyer false.

On a une deux *méthodes* `estInstallé` *: la première vérifie qu'un logiciel* dont on fourni le nom *est installé ; la seconde, qu'on vous demande d'écrire, vérifie qu'un logiciel dont on connaît le nom et la version est installé.*

```
    public boolean estInstalle(Logiciel l){
       // la méthode m.get(k) de Map renvoie
       // l'objet enregistré pour la clef k s'il y en a un,
       // ou null sinon... On utilise cette particularité.
       // alternative: utiliser installes.containsKey(k).
       // mais on devra appeler get de toute manière.
       Logiciel trouvé = installes.get(l.getNom());
       return (
```

```
            trouvé != null
        && l.getVersion() == trouvé.getVersion());
    }
```

*Premier point, important dans ce genre de questions : bien utiliser les collections. On peut toujours parcourir une collection, mais quand ça n'est pas nécessaire, on attend de vous que vous sachiez bien utiliser ses particularités. Par exemple, ici, on a une map. Du coup, on peut facilement avoir accès à un logiciel à partir de son nom. Le code qu'on va écrire est **beaucoup** plus efficace qu'un parcours.*

Remarquez la ligne du return, et en particulier l'ordre des vérifications, qui est important : on ne peut tester le numéro de version de `trouvé` que si `trouvé` n'est pas nul ! on peut aussi faire plus simple (mais moins élégant) :

```
public boolean estInstalle(Logiciel l){
    // la méthode m.get(k) de Map renvoie
    // l'objet enregistré pour la clef k s'il y en a un,
    // ou null sinon... On utilise cette particularité.
    // alternative: utiliser installes.containsKey(k).
    // mais on devra appeler get de toute manière.
    Logiciel trouvé = installes.get(l.getNom());
    if (trouvé == null)
      return false;
    else
      return l.getVersion() == trouvé.getVersion();
}
```

Question 1.2 *(2 points)*

Dans `InstallMachine`, ajoutez la méthode `depNonInstallees(Logiciel p)` qui retourne un `Set<Logiciel>` correspondant aux logiciels se trouvant dans la liste de dépendances de p **et qui par ailleurs ne sont pas installés** sur la machine. Supposons que k soit dans les dépendances de p : s'il est déjà installé (avec la version qui convient), k ne doit pas figurer dans le résultat. Si k n'est pas installé, ou si k est installé mais dans une autre version, il doit figurer au résultat.

L'idée la plus simple est ici de parcourir les dépendances du logiciel `l` et d'ajouter toutes celles qui ne sont pas installées au résultat de la fonction.

```
public Set<Logiciel> depNonInstallees(Logiciel l){
    Set<Logiciel> manque = new HashSet<Logiciel>();
    for (Logiciel d: l.getDependances()){
        if (! estInstalle(d))
          manque.add(d);
    }
    return manque;
}
```

*Autre solution : les dépendances non installées, ce sont les dépendances, **moins** les logiciels installés. On utilise `removeAll` (attention, on aura le problème expliqué dans les dernières questions, mais on s'en fiche par hypothèse.)*

```
public Set<Logiciel> depNonInstallees(Logiciel l){
    Set<Logiciel> logicielsNecessaires= l.getDependances();
    logicielsNecessaires.removeAll(installes.values());
    return logicielsNecessaires;
}
```

*Un point important pour cette solution est que `getDependances()` renvoie une **copie** des dépendances. C'est très important de le remarquer, car votre méthode va modifier le contenu de la collection que lui renvoie `getDependances()`. Typiquement, la javadoc de `getDependances()` devrait bien préciser ce point.*

La solution est très élégante, et si c'est celle que vous avez trouvée, félicitation... Ceci dit, elle est probablement moins efficace que la solution ci-dessus. `removeAll` parcourt la collection la plus petite des deux, et utilise `contains` sur l'autre pour savoir s'il faut ou non garder les éléments. Le problème est que `values()` n'est pas un `Set`, et que sa méthode `contains` sera donc peu efficace.

Question 1.3 *(1.5 points)*

Dans la classe `InstallMachine` complétez la méthode `installer(Logiciel l)`. Note : un logiciel pourra être installé uniquement si ce logiciel (même nom, même numéro de version) n'est pas déjà installé, et si toutes ses dépendances sont déjà installées. Dans les autres cas, le logiciel ne sera pas ajouté aux logiciels installés de la machine et la méthode renverra false.

```
public boolean installer(Logiciel l){
    if (estInstalle(l)){
        return false;
    }
    Set<Logiciel> manque = depNonInstallees(l);
    if (manque.isEmpty()){
        installes.put(l.getNom(), l);
        return true;
    } else {
        return false;
    }
 }
```

À la place de `isEmpty` on peut faire `if (manque.size() == 0)`

Question 1.4 *(2 points)*

Complétez les 4 tests JUNIT suivants, qui doivent permetre de vérifier le bon comportement de la méthode `installer`. Vos tests doivent correspondre à des cas différents de cette méthode.

Il y a beaucoup de solutions ici, d'autant qu'on peut imaginer bien plus de 4 tests... Attention : ils doivent être différents dans ce qu'il vérifient !

Vous supposerez que les déclarations suivantes figurent dans votre fichier de tests :

```
public class LogicielTest {
        public static Logiciel jdk6 = new Logiciel("JDK",6);
        public static Logiciel jdk8 = new Logiciel("JDK",8);
        public static Logiciel netbeans = new Logiciel("netbeans",4);
```

```
    public static Logiciel x11 = new Logiciel("X11",7);


    @BeforeClass  // CECI s'execute avant de commencer l'ensemble des tests
    public static void avantTous() {
        netbeans.ajoutDependance(jdk6);
        netbeans.ajoutDependance(x11);
    }
    @Test
     public void test5() {
        InstallMachine ins = new InstallMachine();
        boolean res = ins.installer(x11);
        // completer...
     }
     @Test
      public void test6() {
        InstallMachine ins = new InstallMachine();
        boolean res = ins.installer(netbeans);
        // completer...
     }
     @Test
      public void test7() {
        InstallMachine ins = new InstallMachine();
        // completer
      }
      @Test
      public void test8() {
        InstallMachine ins = new InstallMachine();
        // completer
      }
}
```

```
@Test
// L'installation d'un logiciel sans dépendances se déroule
// bien :
public void test5() {
        InstallMachine ins = new InstallMachine();
        boolean res = ins.installer(x11);
        assertTrue(res);
        assertTrue(ins.estInstalle(x11));
}

@Test
// L'installation d'un logiciel échoue si ses dépendances manquent
// En "vrai", j'aurais tendance à essayer aussi avec une seule
// des dépendances installées.
public void test6() {
        InstallMachine ins = new InstallMachine();
        boolean res = ins.installer(netbeans);
        assertFalse(res);
        assertFalse(ins.estInstalle(netbeans));
```

```
    }

    @Test
    // On teste le cas où on a une mauvaise version d'une des dépendances
    public void test7() {
            InstallMachine ins = new InstallMachine();
            ins.installer(jdk8);
            ins.installer(x11);
            boolean res = ins.installer(netbeans);
            assertFalse(res);
            assertFalse(ins.estInstalle(netbeans));
    }

    @Test
    // et enfin... test de l'installation d'un logiciel avec
    // dépendances
    // quand celles-ci sont correctes.
    public void test8() {
            InstallMachine ins = new InstallMachine();
            ins.installer(jdk6);
            ins.installer(x11);
            boolean res = ins.installer(netbeans);
            assertTrue(res);
            assertTrue(ins.estInstalle(netbeans));
    }
}
```

On admettait aussi des tests plus simples, qui vérifient simplement la valeur de `res`, et pas que `estInstalle()` renvoyait vrai.

Autres tests possibles : vérifier que si l'on installe deux fois le même logiciel, la seconde installation échoue ; vérification de l'échec de l'installation de netbeans si une seule des dépendances est installée. Vérifier qu'après l'installation d'un logiciel, les logiciels précédemment installés le restent (ça n'est pas explicite dans nos spécifications, mais ça ne dispense pas de le tester)...

Question 1.5 *(1.5 points)*

Dans cette question, on n'est pas certain du respect de l'invariant. Pour chaque test JUNIT ci-dessous, indiquez s'il réussit ou s'il échoue, et expliquez la raison de la réussite ou de l'échec, en une ligne maximum. **Attention : tout réponse non justifié ne sera pas notée**.

```
@Test
 public void test2() {
        Logiciel a = new Logiciel("A", 0);
        Logiciel b0 = new Logiciel("B", 0);
        Logiciel b00 = new Logiciel("B", 0);
        assertTrue(a.ajoutDependance(b0));
        assertFalse(a.ajoutDependance(b00));
}
@Test
 public void test3() {
     Logiciel a = new Logiciel("A", 0);
```

```
        Logiciel b0 = new Logiciel("B", 0);
        Logiciel b1 = new Logiciel("B", 1);
        a.ajoutDependance(b0);
        a.ajoutDependance(b1);
        assertTrue(a.dependDe(b0));
        assertTrue(a.dependDe(b1));
}
@Test
 public void test4() {
     Logiciel a = new Logiciel("A", 0);
     Logiciel b0 = new Logiciel("B", 0);
     Logiciel b00 = new Logiciel("B", 0);
     a.ajoutDependance(b0);
     assertTrue(a.dependDe(b00));
}
```

Lors de l'exécution de ces tests, les ajouts sur l'ensemble `dependances` se font uniquement si aucun objet d'adresse identique n'est pas déjà dans l'ensemble.

1. test2 échoue, car on teste l'appartenance d'un objet d'adresse distincte de celui qui appartient à l'ensemble.
2. test3 réussit car on a ajouté deux objets distincts, et on teste avec **ces mêmes objets.**
3. test4 échoue car le test `assertTrue(a.dependDe(b00))` se fait sur un objet distinct de celui dans l'ensemble.

Question 1.6 *(2 points)*

Si d'après vous, le code des classes ne respecte pas l'invariant, expliquez ce qu'il faut faire pour résoudre le problème (le code exact n'est pas demandé). Si au contraire, vous pensez qu'il le respecte, justifiez votre affirmation (2 lignes maximum).

La classe `Logiciel` ne redéfinit pas hashCode et equals. En conséquence, la méthode `equals` utilisée est celle de `Object`, qui compare les adresses.

Pour obtenir le comportement souhaité dans les tests précédents, il faudrait redéfinir `equals` et `hashCode` pour qu'ils comparent les noms et les versions des logiciels (et éventuellement leurs dépendances).

La remarque qui suit sort un peu du programme ; si vous la trouvez trop complexe, ne vous formalisez pas, vous pouvez temporairement l'ignorer... en revanche, elle a son importance pratique.

Remarque : ne redéfinissez pas ***systématiquement*** *`equals` et `hashCode`. Vous pouvez par exemple décider de ne conserver en mémoire* qu'un seul *objet représentant une version d'un logiciel donné. C'est plus sûr : si vous avez plusieurs objets, ils peuvent avoir par exemple des dépendances différentes...*

Pour cela, il suffit d'avoir un catalogue ou un ensemble des logiciels qui vous permettra d'assurer leur unicité. D'un autre côté, pour récupérer un objet « logiciel » dans ce catalogue, on aura besoin d'un couple `(nom, numéro de version)`...

```
    @Override
    public int hashCode() {
     return nom.hashCode() + 31 * version;
    }
```

```java
@Override
public boolean equals(Object obj) {
  if (obj instanceof Logiciel) {
    Logiciel autre = (Logiciel) obj;
    return this.nom.equals(autre.nom) && this.version == autre.version;
  } else {
    return false;
  }
}
```

Exercice 2 *Entrées/Sorties (7 points)*

Question 2.1 *3 points*

Écrire le code de la procédure suivante :

```
public static void copierLignes(Reader r, Writer w,
        int pos, int nombreLignes)
  throws IOException{
}
```

La fonction doit recopier sur w les lignes de texte lues dans r, en copiant `nombreLignes` lignes, en commençant à la ligne `pos`.

- la numérotation des lignes commence à 0 ;
- Formellement, on doit copier toute ligne dont le numéro est compris entre pos (inclus) et pos+ nombreLignes (exclu).

Donc, si pos=0 et nombreLignes=10, la procédure copiera les 10 premières lignes lues.

Si le flux lu sur reader n'est pas assez long, par exemple, s'il comporte 6 lignes en tout et qu'on a demandé pos=4 et nombreLignes=8), ça ne sera pas considéré comme une erreur. Dans ce cas précis, on se contentera de copier les lignes 4 et 5 (comme la numérotation commence à 0, il n'y a pas de ligne 6). De même, si pos est supérieur ou égal au nombre de lignes lisibles sur r, w sera le fichier vide.

Première question à se poser : quelles classes doit-on utiliser. Dans le « pire » cas, on peut toujours se débrouiller avec un simple `Reader`. Mais il y a souvent plus simple ; si vous vous apercevez que la solution que vous avez choisie vous emmène vers du code compliqué, prenez deux minutes pour réfléchir et voir si vous avez fait le bon choix !

Ici, on parle de lignes, dont le contenu exact ne nous intéresse pas plus que ça... On va donc utiliser un `BufferedReader`.

Une première version, assez « naturelle », ça correspond probablement à la manière dont vous avez vu l'exercice : on lit les pos premières lignes sans rien faire d'autre, puis on lit les lignes demandées en recopiant le texte.

```java
public static void copierLignes(Reader r, Writer w,
                    int pos, int nombreLignes)
        throws IOException{
  BufferedReader buff= new BufferedReader(r);
  int i= 0; // numéro de la ligne courante.
  String l= r.readLine();
  // On lit les pos premières lignes
```

```
  // (attention à la fin de fichier !!!)
  while (i < pos && l != null) {
    i++;
    l= r.readLine();
  }
  // On lit les nombreLignes lignes suivantes
  // (si possible)
  // au départ, si l n'est pas null,
  // il pointe vers la ligne numéro pos
  i= 0; // remise du compteur à 0, on va compter
        // les lignes à écrire.
  while (i < nombreLignes && l != null) {
    w.write(l); on copie la ligne
    w.write('\n'); on écrit un saut de ligne
    i++; on augmente le numéro de ligne
    l= r.readLine(); on passe à la ligne suivante
  }
  buff.close();
}
```

Autre approche, en prenant l'idée suivante : on lit tout le fichier, et pour chaque ligne, on décide si oui ou non on l'affiche. Ça simplifie en particulier le problème de savoir pourquoi on s'arrête.

On peut améliorer ce code en s'arrêtant quand on arrive à `pos+nombreLignes`.

```
public static void copierLignes(Reader r, Writer w,
                        int pos, int nombreLignes)
        throws IOException{
  BufferedReader buff= new BufferedReader(r);
  int i= 0; // numéro de la ligne courante.
  String l= r.readLine();
  while (l != null) {
    // si on est entre les lignes à copier, copier !
    if (pos <= i && i < pos + nombreLignes) {
      w.write(l); on copie la ligne
      w.write('\n'); on écrit un saut de ligne
    }
    i++;
    l= r.readLine();
  }
  buff.close();
}
```

Question 2.2 *4 points*

Dans un logiciel de blog, on a décidé que les rédacteurs pourraient taper des fichiers textes simplifiés au lieu de html. Dans le format qu'on a défini, un lien entre une page et un site web est donnée de la manière suivante :

```
[ LABEL DU LIEN | URL DU LIEN ]
```

Comme par exemple :

```
voir le [cours de NFA035|http://www.cnam.fr/nfa035]
pour plus de détails
```

Le label du lien et son URL sont considérés comme du texte quelconque, avec comme seule condition que le label ne peut pas contenir le caractère '|' et que l'URL n'a pas le droit de contenir le caractère ']'.

On considère la classe java (que vous n'avez pas à écrire).

```
class Lien {
  private String label, url;
  public Lien(String label, String url) {...}
  public String getLabel() {...}
  public String getUrl() {...}
}
```

Écrivez la méthode

```
public static List<Lien> listerLiens(Reader r) throws IOException {
...
}
```

Qui crée et retourne la liste des liens trouvés dans le flux lu par `r`.

Vous supposerez que le fichier est correctement écrit (on ne vous demande pas de gérer les erreurs).

Ici aussi, la première décision est de choisir la bonne classe... on a un fichier structuré, donc le `StreamTokenizer` peut vous tenter... mais en réalité, on n'utilise pas ***du tout*** *la notion de mot... Les seuls cas importants sont les caractères `[`, `]` et `|`. Du coup, on réalise que la lecture* caractère par caractère *est sans doute la meilleure approche.*

La remarque « Vous supposerez que le fichier est correctement écrit (on ne vous demande pas de gérer les erreurs). *» simplifie considérablement le code : elle vous permet de décider que quand on rencontre un « [», on est* ***sûr*** *de rencontrer un « | » puis un «] ». D'où le code :*

```
public static List<Lien> listerLiens(Reader r) throws IOException {
  List<Lien> res= new ArrayList<>();
  int c; // attention au type !!! int.
  while ((c= r.read()) != -1) {
    if (c== '[') {
      String label="";
      String url="";
      // pas de test à -1 à cause de l'hypothèse ci-dessus !
      while ((c= r.read()) != '|') {
        label+= (char)c;
      }
      // actuellement, c=='|'
      while ((c= r.read()) != ']') {
        url+= (char)c;
      }
      res.add(new Lien(label, url));
    }
    // sinon, rien, on avance tout simplement...
  }
  return res;
}
```

Si on craint d'avoir des erreurs (des liens non fermés par exemple), il faut ajouter à tous les while un test vérifiant que c est différent de -1 :

```
while ((c= r.read()) != '|' && c != -1) {
...
}
```

Une seconde solution, plus « tout terrain », est de raisonner en terme d'états. *C'est un peu plus complexe, et la première version répond très bien à l'énoncé. Nous donnons cette solution à titre pédagogique (évidemment, si c'est ce que vous avez écrit, c'est très bien).*

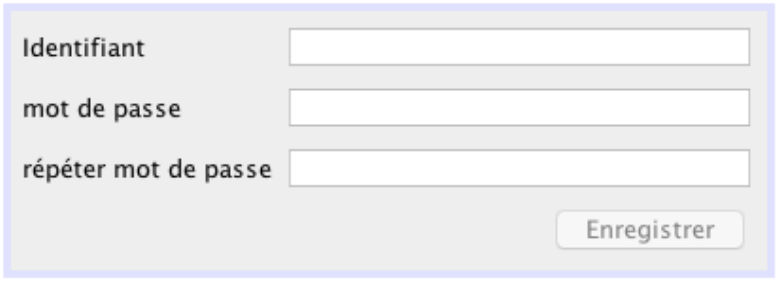
```
public static List<Lien> listerLiens(Reader r) throws IOException {
  // On a trois états, qu'on peut identifier par trois valeurs
  // On utilise des constantes pour améliorer la lisibilité du code.
  final int DANS_TEXTE=0;
  final int DANS_LABEL=1;
  final int DANS_URL=2;

  List<Lien> res= new ArrayList<>();
  int etat= DANS_TEXTE;
  int c; // attention au type !!! int.
  String label="";
  String url="";
  while ((c= r.read()) != -1) {
    if (etat == DANS_LABEL) {
      if (c == '|') {
        etat= DANS_URL;
      } else {
        label+= (char) c; // essayez sans le cast pour voir.
      }
    // attention, ligne suivante, si on supprime le "else"
    // ça buggue: pourquoi ? (c'est une question pour vous,
    // moi je sais !
    } else if (etat == DANS_URL) {
      if (c == ']') {
        res.add(new Lien(label,url));
        label="";
        url="";
        etat= DANS_TEXTE; // manquait ds première version de la correction.
      } else {
        url+= (char)c;
      }
    } else {
      // A priori, etat vaut DANS_TEXTE !
      if (c == '[') {
        etat= DANS_LABEL;
      }
    }
  return res;
}
```

Remarque finale : au lieu de construire label et url par concaténation avec des String, il vaut mieux utiliser une classe prévue pour ça : `StringBuilder`.

Exercice 3 *Swing (3 points)*

On considère l'interface graphique, qui est supposée servir à sauvegarder les informations d'un nouvel utilisateur.



Comme vous le voyez, le bouton pour enregistrer le nouvel utilisateur est désactivé (on l'a configuré avec `bouton.setEnabled(false)`. Pour le réactiver, il faut appeler `bouton.setEnabled(true)`.

On vous demande d'implémenter le comportement suivant :

— le bouton ne doit être activé **que** si l'identifiant et les mot de passe ne sont pas vides, et que d'autre part, les deux mots de passe sont égaux (pour simplifier, on considèrera que les deux mots de passe sont édités par des `JTextField`). L'activation (ou la désactivation) du bouton doit se faire en cours de frappe : dès que le contenu des champs texte est correct, le bouton doit s'activer.

— quand on presse le bouton, il doit appeler la méthode `sauverUtilisateur()`, qu'on ne vous demande pas d'écrire.

Complétez le code suivant, en écrivant la méthode `activer()` (et éventuellement les classes et méthodes auxiliaires dont vous avez besoin)

Vous pouvez ajouter des méthodes à la classe.

```
public class CreateurUtilisateur {
        private JTextField identifiantField= new JTextField(10);
        private JTextField motDePasse1Field= new JTextField(10);
        private JTextField motDePasse2Field= new JTextField(10);
        private JButton bouton= new JButton("Enregistrer");

        public CreateurUtilisateur() {
                mettreEnPage();
                activer();
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
        }

        private void mettreEnPage() {
          // ne pas écrire...
        }

        private void activer() {
                // met en place les divers listeners...
                // À ÉCRIRE !!!

        }
```

```
        /**
         * méthode auxiliaire que vous pouvez utiliser.
         *
         * la méthode est déjà écrite, on ne vous demande
         * pas de l'écrire
         */
        public void sauverUtilisateur(String identifiant, String motDePasse) {
                // NE PAS ÉCRIRE...
        }

}
```

Solution : on met en place un action listener pour le bouton (on va utiliser les lambda pour aller plus vite. Pour la vérification des champs « à la volée » on va utiliser la technique vue en cours, en attachant un DocumentListener aux divers champs.

Pour créer le DocumentListener, on peut utiliser une classe interne comme ci-dessous. On ne peut pas utiliser de lambda (car un DocumentListener a plusieurs méthodes) ; on peut aussi utiliser un `EventHandler` *(assez pratique, toutes les méthodes feront la même chose).*

```
public class CreateurUtilisateur {
 private JTextField identifiantField= new JTextField(10);
 private JTextField motDePasse1Field= new JTextField(10);
 private JTextField motDePasse2Field= new JTextField(10);
 private JButton bouton= new JButton("Enregistrer");

 // classe interne: le documentListener:
 private class MyDocListener implements DocumentListener{
   public void changedUpdate(DocumentEvent e) { /* RIEN */}
   public void insertUpdate(DocumentEvent e) { verifBouton();}
   public void removeUpdate(DocumentEvent e) { verifBouton();}
 }

 private MyDocListener docListener= new MyDocListener();

 public CreateurUtilisateur() {
   mettreEnPage();
   activer();
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setVisible(true);
 }

 private void mettreEnPage() {
   // ne pas écrire...
 }

 private void verifBouton() {
  boolean ok=
        ! identifiantField.getText().isEmpty()
        && ! motDePasse1Field.getText().isEmpty()
        && motDePasse1Field.getText().equals(
                  motDePasse2Field.getText());
```

```
    bouton.setEnabled(ok);
  }

  private void activer() {
    bouton.addActionListener((e) -> sauver());
    identifiantField
      .getDocument().addDocumentListener(docListener);
    motDePasse1Field
      .getDocument().addDocumentListener(docListener);
    motDePasse2Field
      .getDocument().addDocumentListener(docListener);
    // On désactive le bouton:
    verifBouton();
  }

  // Méthode auxiliaire utilisée par notre code...
  // normalement appelable uniquement si tout est bon.
  // On peut éventuellement être plus prudent que ça.
  private void sauver() {
    sauverUtilisateur(identifiantField.getText(),
                      motDePasseField.getText());
  }

  /**
  * méthode auxiliaire que vous pouvez utiliser.
  *
  * la méthode est déjà écrite, on ne vous demande
  * pas de l'écrire
  */
  public void sauverUtilisateur(String identifiant, String motDePasse) {
    // NE PAS ÉCRIRE...
  }
}
```