Examen (avec document)
Corrigé

Préambule : Répondre de manière concise et précise aux questions. Ne pas mettre de commentaires de documentation sauf s'ils sont nécessaires à la compréhension.

Il est conseillé de répondre directement dans le sujet quand c'est possible. Sinon, il est conseillé de mettre une marque sur le sujet (par exemple une lettre majuscule : A, B, C, etc) et d'indiquer sur la copie la marque avec le texte (ou le code) associé.

Les exercices sont relativement indépendants.

Barème indicatif :

exercice	1	2	3	4	5
points	3	3	4	4	6

Exercice 1 Indiquer au moins trois patrons de conception utilisés dans Swing en indiquant clairement où ils apparaissent dans Swing.

Solution :

- Composite pour Container/Component
- Stratégie pour Container/LayoutManager
- Adapteur pour les XAdapter (qui réalise les XListener, par exemple MouseAdapter)
- Décorateur pour JScrolPane
- MVC pour les composants graphiques
- Fabrique abstraite pour les look-and-feel
- ...

Gestion de préférences

L'objectif de ces exercices est de proposer un moyen de définir des préférences. Les préférences permettent de paramétrer une application en associant une valeur à un nom. L'exercice 2 définit la classe `Preference`. L'exercice 3 permet de sauvegarder les préférences au format XML. L'exercice 4 permet d'extraire les préférences à partir d'une classe Java. Enfin, l'exercice 5 propose une application Swing pour saisir les valeurs des préférences.

Exercice 2 : La classe Preference

On considère qu'une préférence est définie par un nom, un type, une description et une valeur. La description peut ne pas être donnée (**`null`**). La valeur initiale est indéterminée (**`null`**) et pourra donc être positionnée ultérieurement.

Le type permet de connaître le type de l'information et est utilisé pour savoir comment initialiser la valeur à partir d'une chaîne de caractères. Le principe est d'utiliser la méthode `valueOf` prenant en paramètre un `String` ou, à défaut, un `Object`. Cette méthode doit bien sûr exister sur le type considéré. Par exemple, les classes `Integer`, `Boolean` ou `Double` définissent la méthode `valueOf(String)` qui retourne un élément du type considéré, construit à partir d'une chaîne de caractères. La classe `String` définit la méthode `valueOf(Object)` qui retourne la représentation de l'objet sous la forme d'une chaîne de caractères (en utilisant `Object.toString()`).

```
1 Double d = Double.valueOf("3.14");
2 Integer i = Integer.valueOf("123");
3 Boolean b = Boolean.valuOf("true");
4 String s = String.valueOf(new Date());
```

On suppose que l'on pourra utiliser les types élémentaires (double, int, boolean, etc.) qui seront remplacés par le type enveloppe correspondant (`Double`, `Integer`, `Boolean`, etc.). De même, on pourra utiliser « string » pour désigner la classe `String`.

Le code partiel de la classe `Preference` est donné au listing 1.

2.1 Indiquer quand est exécuté le code qui commence à la ligne 18.
Solution : C'est un initialiseur statique. Il est exécuté quand la classe `Preference` est chargée par la machine virtuelle.

2.2 Compléter le code du constructeur. Lire attentivement sa documentation.
Solution : Voir listing 1

2.3 Compléter le code de `setValeur(String)`. Lire attentivement sa documentation.
Solution : Voir listing 1

Listing 1 – La classe Preference (code partiel)

```
1  import java.lang.reflect.Method;
2  import java.util.*;
3
4  /** Préférence typée avec initialisation grâce à ce type. */
5  public class Preference {
6      private String nom;
7      private String nomType;
8      private Object valeur;
9      private String description;
10
11     private Method valueOf;
12     private Class<?> type;
13         //@ invariant value != null ==>
14         //@     type.getDeclaringClass() == value.getClass();
15
16     static Map<String, Class<?>> typesPredefinis;
17
18     static {
19         typesPredefinis = new HashMap<String, Class<?>>();
20         typesPredefinis.put("string", String.class);
21         typesPredefinis.put("int", Integer.class);
22         typesPredefinis.put("double", Double.class);
23         typesPredefinis.put("float", Float.class);
24         typesPredefinis.put("boolean", Boolean.class);
```

```
25          // XXX cette liste n'est pas complète
26      }
27
28      /** Initialiser cette préférence à partir de son nom et son type.
29       * Notons que pour les types prédéfinis (int, double, etc) on
30       * utilisera les classes enveloppes correspondantes.  Le type <<
31       * string >> correspondra au type java.lang.String.
32       * @param nom le nom de la propriété
33       * @param type le type de cette propriété
34       * @throws IllegalArgumentException si la classe correspondant à
35       * type n'existe pas (ClassNotFoundException) ou si elle ne contient
36       * pas la méthode valueOf adéquate (NoSuchMethodException).
37       */
38      public Preference(String nom, String type) {
39          this.nom = nom;
40          this.nomType = type;
41
42          // Initialiser this.type (la classe qui correspond au type
43          // this.type et this.valueOf sa méthode valueOf qui prend un
44          // String en paramètre, à défaut un Object.
45          Class<?> typePredefini = typesPredefinis.get(type);
46          try {
47              if (typePredefini != null) {
48                  this.type = typePredefini;
49              } else {
50
51                  this.type = Class.forName(type);
52
53              }
54
55              this.valueOf = findValueOfMethod(this.type, String.class);
56              if (this.valueOf == null) {
57                  this.valueOf = findValueOfMethod(this.type, Object.class);
58                  if (this.valueOf == null) {
59                      throw new IllegalArgumentException(
60                              "Type␣sans␣méthode␣valueOf␣:␣" + type);
61                  }
62              }
63
64          } catch (ClassNotFoundException e) {
65              throw new IllegalArgumentException("Type␣inconnu␣:␣" + type, e);
66          }
67      }
68
69
70      private Method findValueOfMethod(Class<?> classe, Class<?> typeParametre) {
71          try {
72              return classe.getMethod("valueOf", typeParametre);
73          } catch (NoSuchMethodException e) {
74              return null;
75          }
76      }
77
78
```

```
79      public Object getValeur() {
80          return this.valeur;
81      }
82
83      /** Mettre à jour la valeur de cette préférence à partir de la
84       * chaîne de caractères valeur.  Cette mis à jour est réalisée au
85       * moyen de la méthode this.valueOf.
86       * @param valeur valeur à utiliser pour l'initialisation
87       * @throws IllegalArgumentException si l'application de la méthode
88       * valueOf signale un problème (InvocationTargetException).
89       */
90      public void setValeur(String valeur) {
91
92          try {   // XXX devrait être vérifié dans le constructeur !
93              System.out.println("Preference = " + this);
94              this.valeur = valueOf.invoke(null, valeur);
95          } catch (java.lang.reflect.InvocationTargetException e) {
96              // Le paramètre était incorrect !
97              throw new IllegalArgumentException(e);
98          } catch (IllegalAccessException e) {
99              throw new IllegalArgumentException("Droits d'accès insuffisants : "
100                     + type + ".valueOf(String) : ", e);
101         }
102         assert this.valeur.getClass() == valueOf.getDeclaringClass();
103
104     }
105
106     /** Initialiser directement la valeur de l'objet.
107      * @param valeur nouvelle valeur de l'objet
108      */
109     public void setValeur(Object valeur) {
110         this.valeur = valeur;
111     }
112
113     public Class<?> getType() {
114         return this.type;
115     }
116
117     public String getNomType() {
118         return this.nomType;
119     }
120
121     public String getNom() {
122         return this.nom;
123     }
124
125     public void setDescription(String description) {
126         this.description = description;
127     }
128
129     public String getDescription() {
130         return this.description;
131     }
132
```

```
133     @Override
134     public String toString() {
135         return "<" + this.nom + ": " + this.type + " = " + this.valeur + ">";
136     }
137
138
139     public static Preference getPreference(Method m) {
140         String nomMethod = m.getName();
141         String nomPref = Character.toLowerCase(nomMethod.charAt(3)) +
142             nomMethod.substring(4);
143         Class<?> type = m.getParameterTypes()[0];
144         return new Preference(nomPref, type.getName());
145     }
146
147     public static Map<String, Preference> getPreferences(String nomClasse) {
148         Map<String, Preference> preferences =
149                 new HashMap<String, Preference>();
150         try {
151             Class<?> classe = Class.forName(nomClasse);
152             for (Method m : classe.getMethods()) {
153                 String nomMethode = m.getName();
154                 Class<?>[] typesParam = m.getParameterTypes();
155                 if (nomMethode.startsWith("set") && typesParam.length == 1) {
156                     System.out.println(m);
157                     Preference p = getPreference(m);
158                     preferences.put(p.getNom(), p);
159                 }
160             }
161         } catch (ClassNotFoundException e) {
162             throw new IllegalArgumentException("Classe inexistante : "
163                     + nomClasse);
164         }
165         return preferences;
166     }
167
168
169 }
```

Exercice 3 : Engendrer une représentation XML des préférences

On souhaite pouvoir écrire une collection de préférences en XML dans un fichier. L'interface suivante est définie dans cet objectif.

```
1 import java.util.Collection;
2 import java.io.OutputStream;
3
4 public interface IPreferencesXML {
5     void genererXML(OutputStream out, Collection<Preference> preferences);
6 }
```

La DTD suivante a été définie pour définir la structure du document XML.

```
1 <?xml version="1.0" encoding="ISO-8859-1" ?>
2
3 <!ELEMENT preferences   (preference*)>
4 <!ELEMENT preference    (description?)>
```

```
<!ATTLIST preference
                nom     ID      #REQUIRED
                type    CDATA   #REQUIRED
                valeur  CDATA   #IMPLIED
>
<!ELEMENT description   (PCDATA)>
```

Voici un exemple de fichier XML conforme à la DTD précédente.

```
<?xml version="1.0" encoding="ISO-8859-1"  standalone="no"?>

<!DOCTYPE preferences SYSTEM "preferences.dtd">

<preferences>
    <preference nom="debug" type="java.lang.Boolean" >
        <description>
            afficher des informations de mise au point
        </description>
    </preference>
    <preference nom="taille" type="int" valeur="10" >
    </preference>
    <preference nom="fichierLog" type="java.lang.String"
                valeur="/tmp/log.txt" >
        <description>
            Le nom du fichier dans lequel les informations de mise au point
            doivent être affichées.
        </description>
    </preference>
</preferences>
```

3.1 Expliquer l'intérêt d'utiliser `OutputStream` comme type du paramètre alors que l'on veut écrire dans un fichier.

Solution : Ceci permet d'écrire dans un fichier (car `FileOutpoutStream` est un sous-type de `OutputStream`) mais aussi dans tout autre type de `OutputStream` : tube, socket, tableau, etc. C'est donc plus général et sans effort supplémentaire puisque déjà disponible dans les API Java.

3.2 Écrire une réalisation `PreferencesXML` qui utilise JDom. On supposera qu'il existe une méthode `ecrire(Document, OutputStream)` qui écrit un document JDom sur un flot de sortie.

Solution :

```
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.output.*;


public class PreferencesXML implements IPreferencesXML {

    public Element getElement(Preference pref) {
        // Construire l'élément
        Element eltPref =  new Element("preference")
                .setAttribute("nom", pref.getNom())
                .setAttribute("type", pref.getNomType());
```

```
14          if (pref.getValeur() != null) {
15              eltPref.setAttribute("valeur", "" + pref.getValeur());
16          }
17
18          // Traiter la description
19          String description = pref.getDescription();
20          if (description != null && description.length() > 0) {
21              eltPref.addContent(new Element("description").setText(description));
22          }
23          return eltPref;
24      }
25
26      public void genererXML(OutputStream out, Collection<Preference> preferences) {
27          Element racine = new Element("preferences");
28          for (Preference p : preferences) {
29              racine.addContent(getElement(p));
30          }
31          Document document = new Document(racine,
32                  new DocType("preferences", "preferences.dtd"));
33
34          ecrire(document, out);
35      }
36
37
38      public void ecrire(Document document, OutputStream out) {
39          try {
40              XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());
41              sortie.output(document, out);
42          } catch (java.io.IOException e) {
43              throw new RuntimeException("Erreur sur écriture des préférences.", e);
44          }
45      }
46
47      public static void main(String[] args) throws Exception {
48          List<Preference> prefs = new ArrayList<Preference>();
49          prefs.add(new Preference("titre", "java.lang.String"));
50          prefs.add(new Preference("nom", "java.lang.String"));
51          prefs.add(new Preference("volume", "java.lang.Double"));
52          prefs.add(new Preference("vivant", "java.lang.Boolean"));
53          prefs.add(new Preference("nombre", "java.lang.Integer"));
54          prefs.get(0).setDescription("Le titre...");
55          prefs.get(2).setValeur("5.25");
56          prefs.get(4).setValeur("11");
57          new PreferencesXML().genererXML(new FileOutputStream("output.xml"), prefs);
58      }
59
60  }
```

Exercice 4 : Construire des préférences à partir d'une classe

On souhaite maintenant pouvoir construire les préférences à partir d'une classe. Les préférences seront stockées dans un tableau associatif (`Map`) dont la clé sera le nom de la préférence et la valeur la préférence elle-même.

On veut définir la méthode suivante dans la classe `Preference` :

```
public static Map<String, Preference> getPreferences(String nomClasse);
```

Le principe est de considérer que tous les modifieurs de la classe (méthodes dont le nom commence par « set » et qui n'ont qu'un seul paramètre) correspondent à une préférence. Le nom de cette préférence est le nom de la méthode [1] et son type est le type de l'unique paramètre.

Voici un exemple d'un telle classe.

```
1 public class Exemple1 {
2     public void setTitre(String v) { }
3     public void setNom(String v) { }
4     public void setVolume(double v) { }
5     public void setVivant(boolean v) { }
6     public void setNombre(int v) { }
7
8     public void setCartesien(double x, double y) { }
9 }
```

La dernière méthode ne sera pas considérée comme une préférence car elle prend deux paramètres.

4.1 Expliquer l'intérêt d'utiliser un tableau associatif (`Map`) plutôt qu'une collection pour stocker les préférences.

Solution : On a un accès direct et efficace à une information grâce à sa clé. Les préférences seront utilisées pour paramétrer l'application. Il faut donc retrouver la valeur d'une préférence à partir de son nom, d'où l'utilisation d'un tableau associatif.

4.2 Écrire la méthode `getPreferences(String)`.

Solution : Voir listing 1

Exercice 5 : Construire une interface graphique

On veut écrire une application Swing pour renseigner les préférences récupérées d'une classe. Une capture est donnée figure 1. L'utilisateur commence par saisir le nom de la classe (ici « Exemple1 ») qui servira à définir les préférences. Ensuite, il clique sur charger. Les préférences récupérées de la classe sont affichées dans la partie centrale de la fenêtre. L'utilisateur peut alors appuyer sur le bouton « MAJ » qui réalise la mise à jour de la collection `preferences` (collection initialement passée en paramètre du constructeur ou positionnée ensuite par `setPreferences`) en fonction des valeurs saisies dans les zones de saisies correspondantes (utilisation de la méthode `Preference.setValeur(String)`). Le bouton XML permet d'engendrer le fichier XML « output.xml » correspondant aux préférences. Enfin, le bouton « Annuler » permet de quitter l'application.

Le code partiel de la classe `PreferencesSwing` est donné ci-dessous.

5.1 Compléter le code du constructeur. Les « ..... » correspondent à une seule instruction.

Solution : Voir le code ci-dessous.

5.2 Compléter le code de la méthode `setPreferences` qui met à jour l'attribut `preferences` et recrée la partie centrale de la fenêtre (pour chaque préférence, un `JLabel` pour le nom, un

1. En fait, le nom de la préférence devrait être obtenu en supprimant « set » et en mettant en minuscule la première lettre. Ceci n'est pas demandé.

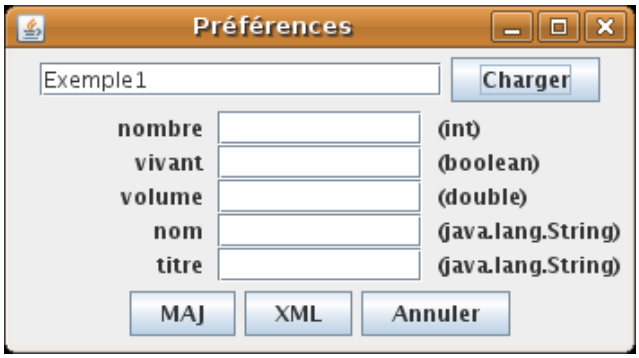

FIGURE 1 – L'application PreferencesSwing, une fois la classe Exemple1 chargée

`JTextField` pour la valeur et une `JLabel` pour le type). Notons que les `JTextField` sont conservées dans l'attribut « zonesDeSaisie ».
Solution : Voir le code ci-dessous.

5.3 Expliquer l'intérêt de l'attribut « zonesDeSaisie » de type `Collection<JTextField>`.
Solution : Dans l'écouteur enregistré auprès du bouton MAJ, on doit pouvoir retrouver les informations saisies par l'utilisateur. On moyen simple est donc de conserver un accès à ces JTextField. C'est l'objectif de `zonesDeSaisie`.
Remarque : Il aurait été possible de servir de la méthode Container.getComponent(int) qui permet de récupérer le ième composant d'un container.

5.4 Rendre actifs les différents boutons.
Solution : Voir le code ci-dessous.

```
1  import java.io.*;
2  import java.util.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class PreferencesSwing extends JFrame {
8
9      private JTextField nomClasse = new JTextField("Exemple1", 20);
10     private JButton bCharger = new JButton("Charger");
11     private JButton bAnnuler = new JButton("Annuler");
12     private JButton bMAJ = new JButton("MAJ");
13     private JButton bXML = new JButton("XML");
14     private Collection<JTextField> zonesDeSaisie;
15     private Collection<Preference> preferences;
16     private JPanel pPrefs;      // le panel contenant les préférences
17
18     public PreferencesSwing(Collection<Preference> preferences) {
19         super("Préférences");
20         Container c = this.getContentPane();
21
22         c.setLayout(new BorderLayout());
23
24         this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
25
26          // Construire la partie supérieure
27          JPanel chargementPreferencePanel = new JPanel();
28
29          chargementPreferencePanel.setLayout(new FlowLayout());
30
31          chargementPreferencePanel.add(nomClasse);
32          chargementPreferencePanel.add(bCharger);
33
34          c.add(chargementPreferencePanel, BorderLayout.NORTH);
35
36
37          // Construire les boutons de commandes inférieurs
38          JPanel boutons = new JPanel();
39
40          boutons.setLayout(new FlowLayout());
41
42          boutons.add(bMAJ);
43          boutons.add(bXML);
44          boutons.add(bAnnuler);
45
46          c.add(boutons, BorderLayout.SOUTH);
47
48
49          // Construire la partie préférences
50          this.pPrefs = new JPanel();
51
52          c.add(pPrefs, BorderLayout.CENTER);
53
54          if (preferences != null) {
55              this.setPreferences(preferences);
56          }
57
58          // Positionner les réactions
59
60          bMAJ.addActionListener(new ActionMAJ());
61          bXML.addActionListener(new ActionXML());
62          bAnnuler.addActionListener(new ActionAnnuler());
63          bCharger.addActionListener(new ActionCharger());
64
65
66          this.pack();
67          this.setVisible(true);
68      }
69
70      public void setPreferences(Collection<Preference> preferences) {
71          this.preferences = preferences;
72          pPrefs.removeAll();     // supprimer tous les composants de pPrefs
73
74          pPrefs.setLayout(new GridLayout(preferences.size(), 3));
75          this.zonesDeSaisie = new ArrayList<JTextField>();
76          for (Preference p : preferences) {
77              pPrefs.add(new JLabel(p.getNom() + "␣␣", SwingConstants.RIGHT));
78              JTextField zone = new JTextField(10);
```

```
79              this.zonesDeSaisie.add(zone);
80              pPrefs.add(zone);
81              pPrefs.add(new JLabel("  (" + p.getNomType() + ")"));
82          }
83
84          this.pack();    // recalculer les dimmensions optimales de la fenêtre
85      }
86
87
88
89      private class ActionCharger implements ActionListener {
90          public void actionPerformed(ActionEvent ev) {
91              String nom = nomClasse.getText();
92              setPreferences(Preference.getPreferences(nom).values());
93          }
94      }
95
96      private class ActionXML implements ActionListener {
97          public void actionPerformed(ActionEvent ev) {
98              try {
99                  FileOutputStream out = new FileOutputStream("output.xml");
100                 new PreferencesXML().genererXML(out, preferences);
101             } catch (java.io.IOException e) {
102                 throw new RuntimeException(e);
103             }
104         }
105
106     }
107
108     private class ActionMAJ implements ActionListener {
109         public void actionPerformed(ActionEvent ev) {
110             Iterator<JTextField> itZoneSaisie = zonesDeSaisie.iterator();
111             for (Preference pref : preferences) {
112                 JTextField zoneSaisie = itZoneSaisie.next();
113                 try {
114                     String valeurTxt = zoneSaisie.getText();
115                     pref.setValeur(valeurTxt);
116                     zoneSaisie.setBackground(pPrefs.getBackground());
117                 } catch (IllegalArgumentException e) {
118                     zoneSaisie.setBackground(Color.RED);
119                 }
120             }
121             System.out.println("Préférence sauvegardées : " + preferences);
122         }
123     }
124
125     private class ActionAnnuler implements ActionListener {
126         public void actionPerformed(ActionEvent e) {
127             System.exit(0);
128         }
129     }
130
131
132     public static void main(String[] args) {
```

```
133
134         EventQueue.invokeLater(new Runnable() {
135             public void run() {
136                 new PreferencesSwing(null);
137             }
138         });
139
140     }
141
142 }
```