Examen (2 heures 30)
Corrigé

Préambule : Lire attentivement les points suivants.

1. La durée de l'épreuve est de 2h30 (3h20 pour les 1/3 temps).
2. Les documents ne sont pas autorisés. Seule une feuille A4 est autorisée qui devra être rendue avec la copie à la fin de l'épreuve. La feuille A4 peut être une photocopie.
3. Répondre de manière concise et précise aux questions. Ne pas mettre de commentaires de documentation sauf s'ils sont nécessaires à la compréhension.
4. Il est conseillé de répondre directement sur le sujet quand c'est possible. Sinon, il est conseillé de mettre une marque sur le sujet (par exemple le numéro de l'exercice suivi d'une lettre majuscule : 1A, 1B, 2A, etc) et d'indiquer sur la copie la marque avec le texte (ou le code) associé.
5. Rendre le sujet avec la copie (mettre son nom sur la première feuille du sujet).

Barème indicatif :

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

Exercice 1 : Comprehension du cours

Répondre de manière concise aux questions suivantes.

1.1. Indiquer ce que nous apprend la trace d'exécution suivante.

```
1 [2]
2 Exception in thread "main" java.lang.ClassCastException: Attempt to insert class
      java.lang.Double element into collection with element type class java.lang.
      Integer
3         at java.util.Collections$CheckedCollection.typeCheck(Collections.java
              :2276)
4         at java.util.Collections$CheckedCollection.add(Collections.java:2319)
5         at Main.main(Main.java:10)
```

Solution : On apprend que le programme n'est pas robuste car il s'est terminé sur une exception non récupérée.

Cette exception est ClassCastException (du paquetage java.lang).

Après les deux-points, on a un message qui explique pourquoi l'exception a été levée (« Attempt to insert... »).

La suivante indique où l'exception a été levée : ligne 2276 du fichier Collections.java, pendant l'exécution de la méthode typeCheck de la classe CheckedCollection, classe locale à Collections.

En lisant depuis le bas, on voit les différentes méthodes appelées : main a été exécutée jusqu'à la ligne 10 où add a été appelée et exécutée jusqu'à la ligne 2319 où typeCheck a été appelée et exécutée jusqu'à la ligne 2276, ligne où l'exception a été levée.

1.2. On distingue généralement deux types d'égalité.

1.2.1. Expliquer la différence entre égalité logique et égalité physique.

Solution : L'égalité physique consiste à vérifier si deux poignées différentes références la même zone mémoire, le même objet.

L'égalité logique consiste à regarder si les deux objets ont même état externe. En général, les deux objets doivent avoir les mêmes valeurs d'attribut

1.2.2. Indiquer comment s'exprime l'égalité logique en Java

Solution : `equals(Object): boolean`, méthode définie dans la classe Object.

1.2.3. Indiquer comment s'exprime l'égalité physique en Java

Solution : `==`

1.3. Considérons la déclaration du listing suivant. Nous supposons qu'elle apparaît dans la classe `Point` vue en cours, TD et TP.

```
1 public class Point {
2     public static final Point origine = new Point(0, 0);
3     ...
4 }
```

1.3.1. Expliquer ce que signifient les mots-clés **`public`**, **`static`** et **`final`**.

Solution :

- public : accessible de tous
- static : caractérise un élément qui est attaché à une classe (ou interface) et pas un objet ce cette classe.
- final : indique que l'on ne peut pas modifier l'élément. L'élément peut être une variable, un paramètre, un attribut, une méthode ou une classe.
  Pour les trois premiers on ne peut pas changer leur valeur.
  Pour la méthode, on ne peut pas la redéfinir dans une sous-classe.
  Pour la classe, on ne peut pas en hériter.

1.3.2. Indiquer, en justifiant la réponse, si on peut être sûr que l'attribut `origine` de la classe `Point` aura toujours pour coordonnées $(0, 0)$.

Solution : Non, c'est la poignée `origine` et non le point attaché qui est déclaré **`final`**. Ainsi, on ne pourra pas affecter. En revanche, on peut lui appliquer des opérations qui modifieront le point attaché. Voici un exemple :

```
1     Point.origine.translater(2, 4);
```

1.4. On considère une interface I qui spécifie la méthode m() et une poignée p non nulle déclarée de type I, expliquer pourquoi on est sûr que p.m() a un sens, c'est-à-dire qu'il y a un code défini pour m(). Indiquer quel est le code qui sera exécuté.

Solution : Puisque p est non nulle, elle a été initialisée avec un objet. Appelons X la classe qui a permis de créer cet objet.

Comme l'affectation a pu avoir lieu entre p et cet objet, c'est que X est une réalisation de I.

X possède donc la méthode m et, puisque qu'une instance a pu être créée, X est donc une classe concrète et donc m est définie. CQFD.

Exercice 2 : Interfaces graphiques avec Swing

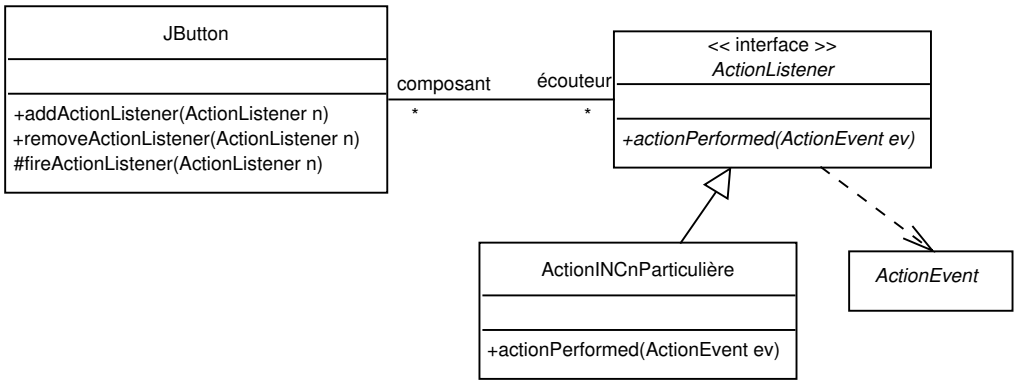
2.1. L'API Swing de Java concernant les interfaces graphiques définit les éléments suivants : ActionListener, ActionEvent, Event, addActionListener et actionPerformed.

2.1.1. Indiquer à quoi correspond chacun de ces éléments.

Solution : ActionListener est l'interface qui spécifie une réaction qui pourra inscrire auprès d'un composant Swing tel qu'un bouton ou une entrée de menu, en fait un élement qui propose la méthode addActionListener. Cette méthode permet d'inscrire une nouvelle réaction auprès d'un composant. ActionListener définit une seule méthode actionPerformed qui correspond donc à la réaction et qui prend en paramètre un ActionEvent, objet qui transporte des informations telles que le composant qui a produit l'événement (getSource), l'action associée (getActionCommand)...

2.1.2. Dessiner un diagramme de classes UML faisant apparaître ces éléments et leurs relations.

Solution :



2.2. On considère une application Swing dont l'apparence est donnée à la figure 1. Elle permet de saisir un code qui s'affiche dans la zone de saisie (partie haute de la fenêtre) au fur et à mesure que l'utilisateur clique sur l'un des boutons correspondant à un chiffre.

Expliquer comment construire la vue de cette application en Swing et donner les principaux éléments du code Java correspondant.

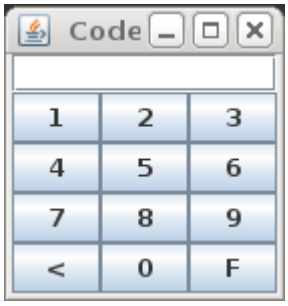


FIGURE 1 – Apparence de l'application

Solution : On remarque qu'il y a deux zones principales dans cette fenêtre :

- En haut, l'afficheur du code en cours de saisie : un seul composant, un JTextField.
- En bas, les boutons qui permettent de saisir un nombre : plusieurs JButton organisés en grille 4 lignes, 3 colonnes.

On utilise donc un gestionnaire de placement de type BorderLayout sur la fenêtre principale et un GridLayout pour le JPanel qui contiendra les boutons.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class Clavier extends JPanel {
6
7      final private JTextField saisie = new JTextField();
8
9      public Clavier(final JFrame fenetrePrincipale) {
10         super(new BorderLayout());
11         saisie.setHorizontalAlignment(JTextField.RIGHT);
12         this.add(saisie, BorderLayout.NORTH);
13         // Définition des boutons
14         JPanel boutons = new JPanel(new GridLayout(4, 3));
15         this.add(boutons, BorderLayout.CENTER);
16         ActionListener cliquer = new ActionCliquer();
17         for (int i = 1; i <= 9; i++) {
18             JButton b = new JButton("" + i);
19             b.addActionListener(cliquer);
20             boutons.add(b);
21         }
22         JButton corriger = new JButton("<");
23         boutons.add(corriger);
24         corriger.addActionListener(new ActionCorriger());
25
26         // Le bouton 0
27         JButton b = new JButton("0");
28         b.addActionListener(cliquer);
29         boutons.add(b);
30
31         // Le bouton Fermer
32         JButton q = new JButton("F");
33         q.addActionListener(new ActionListener() {
34             public void actionPerformed(ActionEvent e) {
35                 fenetrePrincipale.dispose();
36             }
37         });
38         boutons.add(q);
39     }
40
41     private class ActionCliquer implements ActionListener {
42         public void actionPerformed(ActionEvent e) {
43             JButton bouton = (JButton) e.getSource();
44             saisie.setText(saisie.getText() + bouton.getText());
45         }
46     }
47
```

```
48      private class ActionCorriger implements ActionListener {
49          public void actionPerformed(ActionEvent e) {
50              int lg = saisie.getText().length();
51              if (lg > 0) {
52                  saisie.setText(saisie.getText().substring(0, lg -1));
53              }
54          }
55      }
56
57  }
```

2.3. Rendre actifs les boutons. Les chiffres se rajoutent dans la zone de saisie, à la fin. « < » permet de supprimer le dernier caractère saisi. « F » quitte la fenêtre.
Indication : On utilisera un `JTextField` pour représenter la zone de saisie. Cette classe propose les méthodes `setText(String)` et `String getText()` pour manipuler le texte affiché.

La classe `String` fournit une `méthodesubstring(int début,int fin)` qui permet de retourner la sous-chaîne qui commence à l'indice début inclus et se termine à l'indice `fin` exclu.
Solution : Voir le code ci-dessus.

Exercice 3 : Introspection

Écrire une classe principale Main qui affiche toutes les méthodes publiques à deux paramètres de la classe dont le nom est donné sur la ligne de commande.

Par exemple, le résultat de `java Main java.lang.Math` est donné au listing 1.
Solution :

```
1  import java.lang.reflect.*;
2
3  class Main {
4          public static void main(String[] args) throws Exception {
5                  assert args.length == 1;
6                  Class<?> classe = Class.forName(args[0]);
7                  for (Method m : classe.getMethods()) {
8                          if (m.getParameterTypes().length == 2) {
9                                  System.out.println(m);
10                         }
11                 }
12         }
13 }
```

Listing 1 – Résultat de `java Main java.lang.Math`

```
public static double java.lang.Math.atan2(double,double)
public static double java.lang.Math.pow(double,double)
public static int java.lang.Math.min(int,int)
public static long java.lang.Math.min(long,long)
public static float java.lang.Math.min(float,float)
public static double java.lang.Math.min(double,double)
public static int java.lang.Math.max(int,int)
public static long java.lang.Math.max(long,long)
public static float java.lang.Math.max(float,float)
public static double java.lang.Math.max(double,double)
public static double java.lang.Math.scalb(double,int)
public static float java.lang.Math.scalb(float,int)
public static double java.lang.Math.IEEEremainder(double,double)
public static double java.lang.Math.copySign(double,double)
public static float java.lang.Math.copySign(float,float)
public static double java.lang.Math.hypot(double,double)
public static double java.lang.Math.nextAfter(double,double)
public static float java.lang.Math.nextAfter(float,double)
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
```

Exercice 4 : Annuaire

L'objectif de cet exercice est de définir un annuaire simplifié qui permet de conserver des numéros de téléphone. L'interface annuaire (listing 2) spécifie les opérations de l'annuaire de manière minimaliste. Les commentaires de documentation sont incomplets mais un programme de test JUnit est donné (listing 3) pour en préciser le fonctionnement.

4.1. Expliquer ce qu'il faudrait ajouter à l'interface Annuaire pour qu'elle constitue réellement une spécification.

Solution : javadoc (avec les paramètres, les conditions d'application et les effets).
et JML pour formaliser tout ceci.

4.2. Indiquer ce qui justifie que la classe de test AnnuaireTest soit abstraite.

Solution : Car ne connaissant que l'interface Annuaire, on ne sait pas (encore) créer d'objets de ce type.

4.3. Expliquer ce que fait la méthode `getTel()` si le nom fourni ne permet pas de trouver de numéro de téléphone.

Solution : Le programme de test nous indique qu'une exception ContactInconnuException doit

Listing 2 – L'interface Annuaire

```
1 public interface Annuaire {
2
3     /** Enregistrer dans l'annuaire un nom avec son tél... */
4     void ajouter(String nom, String tel);
5
6     /** Obtenir le téléphone d'un contact à partir de son nom... */
7     String getTel(String nom);
8 }
```

Listing 3 – La classe de test AnnuaireTest

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  abstract public class AnnuaireTest {
5      protected Annuaire annuaire;
6
7      @Before public void setup() {
8          this.annuaire = newAnnuaire();
9          this.annuaire.ajouter("XC", "2186");
10         this.annuaire.ajouter("MP", "2185");
11     }
12
13     /** Retourner un annuaire vide. */
14     abstract protected Annuaire newAnnuaire();
15
16     @Test public void testerGetTel() {
17         assertEquals("2185", this.annuaire.getTel("MP"));
18         assertEquals("2186", this.annuaire.getTel("XC"));
19     }
20
21     @Test(expected=ContactInconnuException.class)
22     public void testerGetTelAbsent() {
23         this.annuaire.getTel("MO");
24     }
25
26     @Test public void testerAjouter() {
27         this.annuaire.ajouter("XC", "06...");
28         assertEquals("06...", annuaire.getTel("XC"));
29 }   }
```

être levée.

4.4. Expliquer l'information qu'il y a dans l'annuaire si on ajoute deux numéros de téléphone différents pour un même nom.

Solution : Le dernier numéro de téléphone enregistrée. La méthode enregistrer permet donc soit d'ajouter une nouvelle information, soit de mettre à jour une information existante.

4.5. Écrire la classe `ContactInconnuException`.

Solution : L'exception ne doit pas être contrôlée car elle n'apparaît pas dans la clause **`throws`** de la méthode `getTel()` dans l'interface `Annuaire`.

```
1  public class ContactInconnuException extends RuntimeException {
2      public ContactInconnuException(String message) {
3          super(message);
4      }
5  }
```

4.6. Écrire une réalisation de l'interface `Annuaire` appelée `AnnuaireConcret` en veillant à ce que :

1. l'opération `getTel(String nom)` soit efficace. Elle retourne le téléphone d'une personne à partir du nom de la personne.
2. la méthode toString() affichera l'annuaire dans l'ordre alphabétique des noms. Exemple :

```
MP  2185
XC  2186
```

On choisira avec soin les structures de données à utiliser !

Solution : On utilise un tableau associatif Map et plus précisément TreeMap pour que l'affichage se fasse dans l'ordre alphabétique des clés.

```
1  import java.util.*;
2
3  public class AnnuaireConcret implements Annuaire {
4      private Map<String, String> contacts;
5
6      public AnnuaireConcret() {
7          this.contacts = new TreeMap<>();
8      }
9
10     public void ajouter(String nom, String tel) {
11         contacts.put(nom, tel);
12     }
13
14     public String getTel(String nom) {
15         String tel = contacts.get(nom);
16         if (tel == null) {
17             throw new ContactInconnuException("Contact␣inconnu:␣" + nom);
18         }
19         return tel;
20     }
21
22     @Override public String toString() {
23         String resultat = "";
24         for (Map.Entry<String, String> contact: this.contacts.entrySet()) {
25             resultat += contact.getKey() + "\t" + contact.getValue() + "\n";
26         }
27         return resultat;
```

```
28      }
29
30  }
```

4.7. `AnnuaireConcret` doit réussir les tests de `Annuaire`. Écrire la classe de test qui le fait.
Solution : On s'appuie sur AnnuaireTest en héritant de cette classe et en définissant la méthode newAnnuaire pour qu'elle construise et retourne un AnnuaireConcret.

```
1  public class AnnuaireConcretTest extends AnnuaireTest {
2      protected Annuaire newAnnuaire() {
3          return new AnnuaireConcret();
4      }
5
6      public static void main(String[] args) {
7          org.junit.runner.JUnitCore.main(AnnuaireConcretTest.class.getName());
8      }
9
10 }
```

Exercice 5 : XML

On considère la DTD du listing 4.
5.1. Expliquer la signification de `+`, `ID`, `EMPTY` et `|`.
Solution :

- `+` : l'élément doit apparaître au moins une fois.
- `ID` : l'attribut est un identifiant, sa valeur doit donc être unique
- `EMPTY` : un élément n'a pas de contenu (on peut donc utiliser une balise autofermante).
- `|` : choix entre deux motifs

5.2. Expliquer ce qu'est un document XML bien formé et valide.
Solution : Bien formé : respecte les règles XML (balises bien appairées, attributs, etc.)
Valide : respecte une DTD (ou autre moyen de définir la structure du document)
5.3. Écrire un document XML bien formé et valide pour la DTD du listing 4.
Solution :

```
1  <?xml version="1.0" encoding="UTF-8"  standalone="no"?>
2
3  <!DOCTYPE choses SYSTEM "choses.dtd">
4
5  <choses>
```

Listing 4 – Une DTD

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2
3  <!ELEMENT choses ((truc|chose)+)>
4  <!ELEMENT truc (#PCDATA)>
5  <!ELEMENT chose EMPTY>
6  <!ATTLIST chose
7          a ID #REQUIRED
8          b CDATA #REQUIRED
9          c CDATA #IMPLIED>
```

```
6       <truc>T1</truc>
7       <chose a='id1' b='?' />
8       <chose a='id2' b='?' c='x' />
9   </choses>
```

Exercice 6 : XML

Considérons le code Java du listing 5.

6.1. Donner le document XML qui sera affiché sur la sortie standard par ce programme.

Solution :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <polygone>
3    <point x="1" y="0">O</point>
4    <point x="0" y="1">N</point>
5    <point x="-1" y="0">E</point>
6    <point x="0" y="-1">S</point>
7  </polygone>
```

6.2. Donner la signature de la méthode `setAttribute` utilisée.

Solution :

```
1       Element setAttribute(String, String)
```

6.3. Proposer une DTD pour le document XML sachant qu'un point a nécessairement une abscisse et une ordonnée mais peut également avoir une couleur (information optionnelle).

Solution :

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2
3  <!ELEMENT polygone (point*)>
4  <!ELEMENT point (#PCDATA)>
5  <!ATTLIST point
6          x CDATA #REQUIRED
7          y CDATA #REQUIRED
8          color CDATA #IMPLIED>
```

Listing 5 – La classe Main

```
1  import org.jdom2.*;
2
3  class Main {
4          static Element getPoint(String name, int x, int y) {
5                  Element point = new Element("point");
6                  point.setAttribute("x", "" + x).setAttribute("y", "" + y);
7                  point.addContent(name);
8                  return point;
9          }
10         public static void main(String[] args) {
11             Element racine = new Element("polygone");
12                 racine.addContent(getPoint("O", 1, 0));
13                 racine.addContent(getPoint("N", 0, 1));
14                 racine.addContent(getPoint("E", -1, 0));
15                 racine.addContent(getPoint("S", 0, -1));
16                 Document doc = new Document(racine);
17                 JDOMTools.ecrire(doc, System.out);
18         }
19 }
```