Devoir surveillé de COO

<2014-12-17 Wed 14:00>–<2014-12-17 Wed 17:00>

Durée : 3h

Le polycopié de COO (avec notes personnelles) est autorisé ainsi qu'un dictionnaire de langue (papier ou électronique *dédié*). Tout autre document est interdit.

Les exercices sont indépendants. La clarté des réponses sera prise en compte dans l'évaluation. Soignez la présentation en évitant les ratures ! La javadoc et les tests ne sont à fournir que lorsque cela est explicitement demandé.

À titre purement indicatif, durée estimée et suggérée de traitement des exercices : lecture du sujet : 20 min – exercice 1 : 20 min – exercice 2 : 60 min – exercice 3 : 60 min – relecture : 20 min.

Répondre sur deux copies (et leurs intercalaires) séparées, **copie jaune** pour les exercices 1 et 2, **copie verte** pour l'exercice 3. Rendez une copie de chaque couleur, même vide.

1 Questions de cours (4 points environ)

Répondre à ces questions sur la *copie jaune*.

Les génériques

1. Expliquez ce que sont, dans chaque cas suivant, les éléments A, B et C (classe, interface ou type inconnu, générique ou pas, ...) et la relation entre ces éléments (super classe ou sous classe, qui définit ou utilise un type paramétré, ...). Vous devez être précis. Vous *ne devez pas* expliquer le mécanisme des génériques Java en général, mais vous concentrer sur les 3 cas particuliers ci-dessous.

```
Cas 1 : public class B extends A<C> {}
Cas 2 : public class B<C> extends A {}
Cas 3 : public class B<C> extends A<C> {}
```

Les tests

2. Indiquer une technique permettant d'écrire un test unitaire vérifiant qu'un contrat valable sur un type abstrait (par exemple le type `Cell` du TD 1) est aussi valable sur tous ses sous types (par exemple `TeleportCell` et `JailCell`). Vous pouvez illustrer votre explication en dessinant un diagramme ou en écrivant du code mais *vous devez rester concis*.
3. Indiquer, sans rentrer dans les détails, comment écrire un test *unitaire* d'une méthode `m(B b)` dans une classe `A`. Ce test doit être indépendant du code de `B`.

La réflexion

4. Quel est le type de la valeur retournée par l'instruction suivante ?

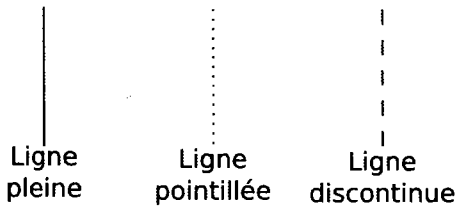
   ```
   "schtroumpf".getClass();
   ```

5. Donnez le code Java permettant de récupérer la super classe de la classe `MyClass`.

2 Conception (8 points environ)

On se propose d'implémenter une application qui réalise des dessins de lignes sur un afficheur. Une ligne se définit par 2 points représentant ses extrémités. Il existe trois types de lignes (voir figure ci-dessous) :

— la ligne pleine (*plain line*) représentée par 1 trait ;
— le ligne pointillée (*dotted line*) représentée par des points régulièrement espacés ;
— le ligne discontinue ou tirets (*dashed line*) représentée par des traits de même longueur et régulièrement espacés.



Ces lignes peuvent être dessinées à l'écran par un afficheur de type *Cairo* ou *Java2D*. Ces afficheurs disposent des méthodes et types suivants (que l'on considère fournis et que vous n'aurez pas à implémenter) :

```
public class CairoCanvas {
  /**
   * Draw a line between points 'p1' and 'p2'. The kind of line is specified
   * by the 'type' parameter.
   */
  public void printStraight(Point p1, Point p2, LineType type) { ... }

  enum LineType {
    PLAIN, DASHED, DOTTED
  }

}
public class Java2DCanvas {
  /**
   * Draw a line between the points 'p1' and 'p2'. The line will be dotted if
   * and only if 'dotted' is true. The line will be dashed if and only if
   * 'dashed' is true. If both 'dotted' and 'dashed' are true, the result is
   * unspecified.
   */
  public void j2dDrawLine(Point p1, Point p2, boolean dotted, boolean dashed) { ... }
}
```

On considère que les 3 types de lignes décrites ci-dessus (pleine, pointillée et discontinue) représentent toutes les lignes dont l'application aura besoin et l'on ne prévoit pas d'en ajouter d'autre. En revanche, dans un avenir proche, on voudra ajouter de nouveaux types d'afficheurs (afficheurs *OpenGL*, *Balloon*, ...).

Vocabulaire

Français	Anglais	Français	Anglais
afficheur	display	dessiner	draw
flèche	arrow	ligne	line
point	point	tête	head
triangulaire	triangular	triple	triple
simple	simple		

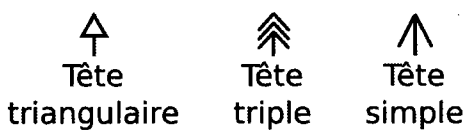
Questions

Répondre à ces questions sur la *copie jaune*. Il est possible de répondre à la question 3 sans avoir réussi la question 2. Le texte de la question 2 peut vous aider à trouver une réponse à la question 1.

1. Proposez une architecture objet pour cette application avec ses types (classes et interfaces), leurs attributs et la (ou les) méthode(s) d'affichage, le tout sous la forme d'un diagramme de classes UML. Votre diagramme devrait permettre d'expliquer les grands principes de l'architecture de cette application. Les classes `CairoCanvas` et `Java2DCanvas` ne doivent pas être modifiées. Vous pouvez compléter votre diagramme par du code Java si besoin. Attention : vous *ne devez pas* écrire l'ensemble du code Java, concentrez-vous sur ce qui vous paraît important pour illustrer votre diagramme.
2. En commençant par la définition de la méthode ci-dessous, écrivez le code des méthodes et classes nécessaires à l'affichage d'une collection de lignes (dont le type exact n'est pas connu à la compilation). Vous devez vous concentrer sur l'afficheur Cairo et *ne pas faire* l'afficheur Java2D. Vous devez écrire le code qui appelle la méthode `printStraight()` mais vous *ne devez pas* implémenter le contenu de cette méthode qui est considérée fournie.

```
public class GUI {
  public void drawLines(Collection<Line> lines, Display display) {
    ...
  }
}
```

3. On souhaite ajouter la notion de flèche à l'application. Une flèche se représente par une ligne et une tête de flèche. Une tête de flèche est un ensemble de lignes. On trouve de nombreux types de têtes de flèche, 3 sont donnés dans la figure ci-dessous. Faites un nouveau diagramme pour que votre architecture prenne en compte les flèches. Concentrez vous sur les différences avec votre diagramme précédent. Vous pouvez compléter votre diagramme par du code Java si besoin. Attention : vous ne *devez pas* écrire l'ensemble du code Java, concentrez-vous sur ce qui vous paraît important pour illustrer votre diagramme. Indiquez aussi comment adapter votre code si nécessaire.



3 Évolution de programme (8 points environ)

Soit le code des classes `StringBasedBooleanEvaluator` et `StringBasedBooleanEvaluatorTest` situé en annexe. De la documentation sur certaines méthodes Java utilisées est également disponible en annexe.

On souhaite faire évoluer ce code pour pouvoir ajouter de nouveaux opérateurs booléens (par exemple `not` et `implies`). On souhaite aussi proposer plusieurs façons d'afficher des expressions booléennes en utilisant les notations préfixée, infixée et postfixée. Avant de faire cela, on envisage de réorganiser l'application pour faciliter ces changements.

Questions

Répondre à ces questions sur la *copie verte*.

1. Écrivez la trace de l'exécution de l'expression `assertEvalTrue("true || false && true");` en indiquant quelles sont les méthodes exécutées et leurs paramètres.
2. Donnez un diagramme de classes UML présentant l'architecture d'un évaluateur d'expressions booléenes facilitant les changement prévus. Vous veillerez à respecter les principes de la programmation objet. Vous pouvez compléter votre diagramme par du code Java si besoin. Attention :

vous ne *devez pas* écrire l'ensemble du code Java, concentrez-vous sur ce qui vous paraît important pour illustrer votre diagramme. De plus, vous ne considérerez que les opérateurs `&&` et `||`, pas les futurs opérateurs `not` et `implies`.

3. Écrivez le code Java permettant de faire fonctionner les tests unitaires fournis, sans les changer. Vous devez pour cela adapter le code de la classe `StringBasedBooleanEvaluator`, et notamment sa méthode `eval()`. Votre nouveau code pourra commencer par construire l'expression booléenne avant de l'évaluer.
4. On souhaite pouvoir afficher les expressions booléennes en utilisant, au choix, la notation préfixée, infixée ou postfixée. Pour l'expression `((a||b)&&c)` (`a`, `b` et `c` valant `true` ou `false`), les trois notations donnent :

   préfixée : `&& || a b c`

   infixée : `( ( a || b ) && c )`

   postfixée : `a b || c &&`

   On considère que le code de ces méthodes est déjà écrit et retourne une chaîne de caractères. Donnez le code des tests unitaires correspondants à la notation préfixée.

4 Annexe : Java API

Arrays.asList()

```
public static <T> List<T> asList(T... a)
```

Returns a fixed-size list backed by the specified array. (Changes to the returned list "write through" to the array.) This method acts as bridge between array-based and collection-based APIs, in combination with Collection.toArray(). The returned list is serializable and implements `RandomAccess`.

This method also provides a convenient way to create a fixed-size list initialized to contain several elements :

```
List<String> stooges = Arrays.asList("Larry", "Moe", "Curly");
```

— Parameters

a - the array by which the list will be backed

— Returns : a list view of the specified array

String.split()

```
public String[] split(String regex)
```

Splits this string around matches of the given regular expression.
The string `boo:and:foo`, for example, yields the following results with these expressions :

Regex	Result
":"	{ "boo", "and", "foo" }
"o"	{ "b", "", ":and:f" }

— Parameters :

regex - the delimiting regular expression

— Returns : the array of strings computed by splitting this string around matches of the given regular expression

— Throws :

PatternSyntaxException - if the regular expression's syntax is invalid

5 Annexe : Code source de l'exercice 3

```
public class StringBasedBooleanEvaluator {

  public boolean eval(List<String> expressions) {
    if (expressions.size() == 1) {
      return convertFromString(expressions.get(0));
    }

    if (expressions.size() >= 3) {
      boolean leftOperand = convertFromString(expressions.remove(0));
      String operator = expressions.remove(0);
      boolean rightOperand = convertFromString(expressions.remove(0));
      switch (operator) {
      case "&&":
        expressions
            .add(0, convertToString(leftOperand && rightOperand));
        break;
      default:
        expressions
            .add(0, convertToString(leftOperand || rightOperand));
      }
      return eval(expressions);
    }
    return false;
  }

  protected String convertToString(boolean b) {
    if (b) {
      return "true";
    } else {
      return "false";
    }
  }

  protected boolean convertFromString(String expression) {
    switch (expression) {
    case "true":
      return true;
    default:
      return false;
    }
  }

  public boolean evalExpressions(String[] strings) {
    List<String> expressions;
    expressions = new ArrayList<String>(Arrays.asList(strings));
    return eval(expressions);
  }

  public boolean evalExpression(String expression) {
    return evalExpressions(expression.split(" "));
  }
}
```

```
public class StringBasedBooleanEvaluatorTest {

  @Test
  public void testEmpty() {
    assertEvalFalse("");
  }

  @Test
  public void testAtoms() {
    assertEvalTrue("true");
    assertEvalFalse("false");
  }

  @Test
  public void testSimpleConjunction() {
    assertEvalTrue("true && true");
    assertEvalFalse("true && false");
    assertEvalFalse("false && true");
    assertEvalFalse("false && false");
  }

  @Test
  public void testSimpleDisjunction() {
    assertEvalTrue("true || true");
    assertEvalTrue("true || false");
    assertEvalTrue("false || true");
    assertEvalFalse("false || false");
  }

  @Test
  public void testWithMoreThan3Elements() {
    assertEvalTrue("true || false && true");
    assertEvalFalse("true && true && false");
    assertEvalFalse("false || false || false");
  }

  protected void assertEvalTrue(String expression) {
    assertTrue(eval(expression));
  }

  protected void assertEvalFalse(String expression) {
    assertFalse(eval(expression));
  }

  protected boolean eval(String expression) {
    return new StringBasedBooleanEvaluator().evalExpression(expression);
  }
}
```