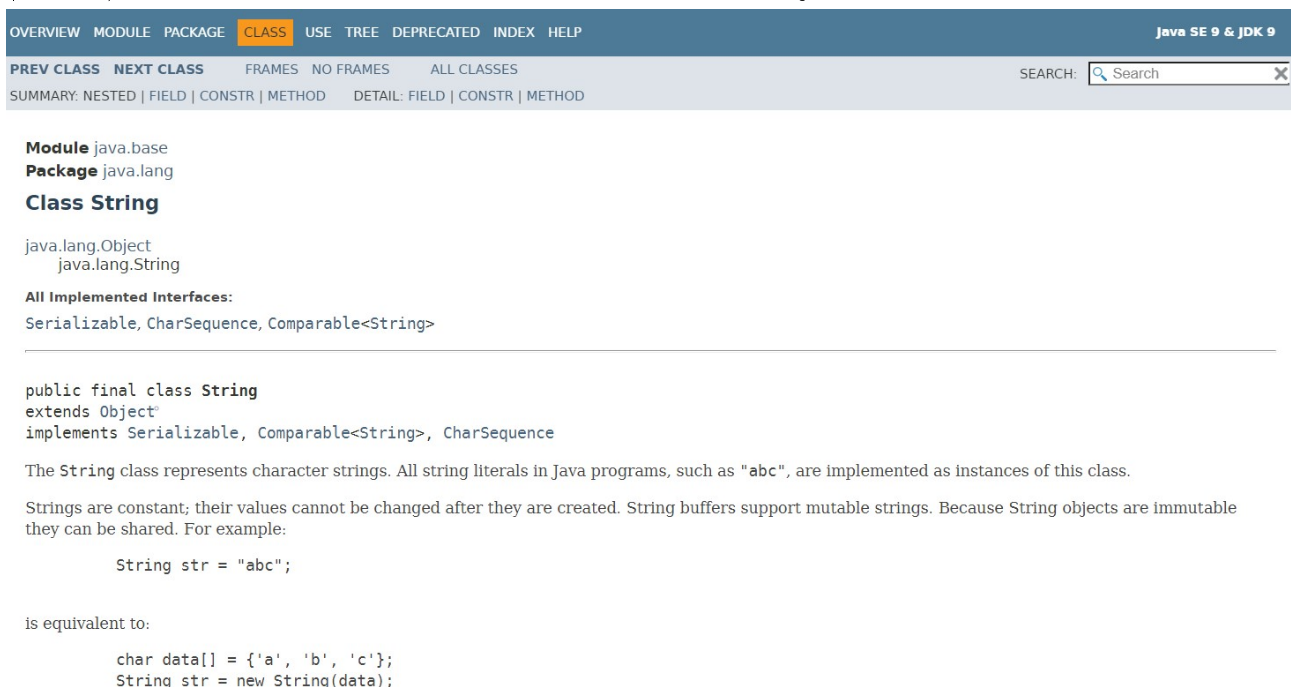
INF103 - Langage JAVA

Contrôle de connaissances (1h30)

4 février 2021

— Le sujet comporte 12 pages.
— Pas de documents autorisés. Téléphones et ordinateurs éteints.
— Le barème est donné à titre indicatif. Il pourra être revu si le correcteur l'estime nécessaire.
— Les réponses aux questions s'écrivent sur le sujet dans les boîtes prévues à cet effet.
— La taille des boîtes n'est pas forcément proportionnelle à la taille des réponses attendues.
— Soignez l'écriture parce qu'un professeur qui a du mal à lire devient de mauvaise humeur :-(
— Si vous constatez une erreur dans l'énoncé, écrivez comment vous la corrigez et continuez votre rédaction.
— Si vous voyez une imprécision dans l'énoncé, écrivez comment vous la précisez et continuez votre rédaction.

1. *(1 point)* On montre le début de la javadoc de la classe `String` :



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java SE 9 & JDK 9

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Est-il possible d'écrire une classe `MyString` qui hérite de `String` afin de rajouter ou de redéfinir vos propres méthodes ? Pourquoi ?

On ne peut pas à cause du mot clé `final` qui interdit que l'on puisse hériter de la classe `String`. [Cela a été fait pour des raisons de sécurité.]

2. Soit le programme de la classe `A` :

```
public class A {
   public void f() { ... }
}
```

Puis voici la classe `B` qui hérite de `A` et redéfinit `f` :

```
public class B extends A {
   @Override
   public void f() { ... }
}
```

(a) *(1 point)* A quoi sert `@Override` ?

(b) *(1 point)* Si on décide d'écrire une méthode `g` dans la classe `B`, comment peut-on y appeler la méthode `f` de la classe `A` ?

> (a) `@Override` est une annotation qui dit au compilateur que la méthode qui suit est une redéfinition d'une méthode héritée. Le compilateur vérifie cela.
>
> (b) On écrit `super.f()`, c'est l'appel à la super-méthode.

3. *(1 point)* Qu'appelle-t-on « encapsulation des données » ?

> L'encapsulation des données signifie que les données des objets, c'est-à-dire les attributs, ne sont pas directement accessibles depuis l'extérieur de l'objet. On ne veut pas savoir comment les objets de la classe sont implémentés. Les attributs doivent être déclarés `private` et on fournit éventuellement des fonctions d'accès (*getters*) et de modifications (*setters*) pour ces attributs.

4. *(1 point)* Citer sans les détailler trois raisons pour encapsuler les données.

> — Valider les valeurs que l'on donne aux attributs.
> — Cohérence et intégrité des structures de données.
> — Cohérence de l'application.
> — Transparence des mises en œuvre des classes.

5. *(2 points)* Qu'appelle-t-on « liaison dynamique de méthode » et « liaison statique de méthode » en programmation O.O. ? Que fait Java dans ce domaine ?

> Si une variable ou un attribut de type référence est déclarée avec un type `T`, elle peut référencer un objet instance de la classe `T` ou bien un objet instance d'une sous-classe de `T`.
>
> Lorsque l'on demande l'exécution d'une méthode, la liaison dynamique de méthode consiste à prendre la méthode correspondante dans la classe d'instanciation de l'objet référencé plutôt que dans la classe de déclaration de la variable.
>
> Si l'on décide de se baser sur la classe de déclaration de la variable, on parle alors de liaison statique.
>
> Java pratique la liaison dynamique de méthode.
>
> Java peut pratiquer la liaison statique de méthode si une méthode est déclarée `final` dans une classe et l'objet qui exécute cette méthode est une instance de la cette classe (optimisation).

6. *(1 point)* Une classe Java peut éventuellement hériter directement d'une seule autre classe et, en même temps, implémenter plusieurs interfaces. Est-ce vrai ?

> Oui.

7. *(1 point)* Voici une classe qui est censée représenter une somme en Euros :

```
public final class Euros {

   private final amount ;

   public int getAmount() {
      return amount ;
   }

   public Euros(int amount)  {
      this.amount = amount ;
   }
}
```

Supposons que l'on ait une variable `e` de type `Euros` et on désire que l'instruction :

```
System.out.print(e) ;
```

affiche la chaîne :

```
<amount> euros
```

où `<amount>` est la valeur de l'attribut `amount` de `e`. Que faut-il faire ?

Il suffit de redéfinir la méthode `toString()` dans la classe `Euros` pour qu'elle fasse le job :

```
@Override
public final String toString() {
      return "" + amount + " euros" ;
}
```

Si on veut être sûr du résultat, il faut mettre le `final`.

8. *(1 point)* Qu'appelle-t-on « surcharge de méthode » en Java ?

C'est quand plusieurs méthodes dans une classe ont le même nom. Elles ont bien sûr des paramètres différents soit en nombre, soit en types, soit les deux.

Exercice de programmation

Comme on vous laisse une grande latitude dans la conception de vos programmes, on vous demande bien expliquer ce que vous programmez, de bien dire dans quelles classes vont vos méthodes, de bien nommer vos variables et vos méthodes suivant la coutume Java, etc.

Un sujet d'examen est composé d'une introduction sous la forme d'une chaîne de caractères puis d'une liste de questions. Les questions peuvent être simples ou complexes :

(a) Une question simple est simplement composée d'une chaîne de caractères qui est la question.

(b) Une question complexe est composée d'une chaîne de caractères, qui sert d'introduction, puis d'une liste de sous-questions qui peuvent être simples ou complexes.

9. *(3 points)* Programmer les classes `ExamSubject`, `SimpleQuestion` et `ComposedQuestion` permettant de représenter respectivement les sujets d'examen, les questions simples et les questions complexes.

Le constructeur de la classe `ExamSubject` doit créer un sujet vide de questions et on programmera une méthode `add` pour y rajouter des questions.

De la même manière, le constructeur de la classe `ComposedQuestion` doit créer une question sans aucune sous-questions et on programmera une méthode `add` pour rajouter des sous-questions.

On n'écrira ni *getters* ni *setters* dans ces classes.

On rappelle quelques méthodes de la classe `ArrayList<E>` du *package* `java.util` :

(a) `boolean add(E e)` : appends the specified element to the end of this list.

(b) `boolean contains(Object o)` : returns `true` if this list contains the specified element.

(c) `boolean remove(Object o)` : removes the first occurrence of `o` from this list, if it is present.

(d) `E remove(int index)` : removes the element at the specified index in this list.

(e) `E get(int index)` : returns the element at the specified index in this list.

(f) `int size()` : returns the number of elements in the list.

Comme les questions peuvent être simples ou complexes, on fera une classe mère de nom `Question` dont `SimpleQuestion` et `ComposedQuestion` seront des filles (des classes qui en héritent). La classe `Question` sera déclarée abstraite afin qu'il n'y ait aucune instance de cette classe.

De cette manière, si nous avons besoin d'une variable qui contienne soit une question simple soit une question complexe, celle-ci sera de type `Question`.

Dans un fichier `Question.java` :

```
public abstract class Question {}
```

Dans un fichier `SimpleQuestion.java` :

```
public final class SimpleQuestion extends Question {

    private final String question ;

    public SimpleQuestion(String question) {
        this.question = question ;
    }

}
```

Dans un fichier `ComposedQuestion.java` :

```
import java.util.ArrayList ;

public final class ComposedQuestion extends Question {

    private final String              introduction ;
    private final ArrayList<Question> sousQuestions ;

    public ComposedQuestion(String introduction) {
        this.introduction  = introduction ;
        this.sousQuestions = new ArrayList<Question>() ;
    }

    public final void add(Question sousQuestion) {
        sousQuestions.add(sousQuestion) ;
    }

}
```

Dans un fichier `SujetExamen.java` :

```
import java.util.ArrayList ;

public final class ExamSubject {

    private final String introduction ;
    private final ArrayList<Question> questions ;

    public ExamSubject(String introduction) {
        this.introduction = introduction ;
        this.questions    = new ArrayList<Question>() ;
    }

    public final void add(Question e) {
        questions.add(e) ;
    }

}
```

10. *(1 point)* Programmer une classe `Main` avec une méthode `main` qui instanciera un sujet avec deux questions simples et une complexe.

Dans un fichier `Main.java` :

```
public final class Main {

    public static void main(String[] args) {

        ExamSubject examSubject = new ExamSubject("Subject INF 103") ;

        examSubject.add(new SimpleQuestion("What is a class ?"));

        examSubject.add(new SimpleQuestion("What is a type ?")); ;

        ComposedQuestion qc
            = new ComposedQuestion("Please program the following:") ;
        qc.add(new SimpleQuestion("The method add.")) ;
        qc.add(new SimpleQuestion("The method remove")) ;
        examSubject.add(qc);
    }

}
```

11. *(3 points)* On désire une méthode de signature `public void print()` de la classe `ExamSubject`. Cette méthode doit afficher les questions du sujet avec leurs numéros qui sont calculés comme suit :

    — Les questions de premier niveau, complexe ou simple, du sujet sont simplement numérotés dans l'ordre d'apparition 1., 2., 3., etc. et on fait suivre le numéro du texte de la question.

    — Les sous-questions d'une question complexe sont numérotés dans l'ordre d'apparition avec un préfixe qui est le numéro de la question complexe englobante. Exemple : si la question 2 est une question complexe, les sous-questions seront numérotées 2.1., 2.2., 2.3., etc. On fait suivre le numéro du texte de la question. Bien entendu, comme il peut y avoir des questions complexes à l'intérieur de questions complexes, cela doit pouvoir se faire à n'importe quel niveau.

    — Dans le cas de `ExamSubject` et de `ComposedQuestion`, on n'oubliera pas d'imprimer la chaîne de caractères d'introduction.

    Programmer la méthode `print()`.

Dans la classe `Question`, on ajoute la méthode abstraite `print` à 2 arguments ainsi que la méthode `print` à 3 arguments qui sera la même pour tous les types ce questions :

```
public abstract class Question {

   /** Print a question.
     * @param prefix the prefix to print before the number,
     *               null if there is no prefix
     * @param number the number of the question
     */
   public abstract void print(String prefix, int number) ;

   /** Print the full number of a question followed by the question
     * @param prefix   the prefix to print before the number,
     *                 null if there is no prefix
     * @param number   the number of the question
     * @param question the question to be printed
     */
   public final void print(String prefix, int number, String question) {
      if (prefix != null) {
         System.out.print(prefix) ;
         System.out.print('.');
      }
      System.out.print(number);
      System.out.print(". ");
      System.out.println(question);
   }
}
```

Dans la classe `ExamSubject`, on ajoute la méthode `print` qui imprime l'introduction du sujet suivi par l'impression des questions avec un préfixe qui est `null` au premier niveau :

```
    public final void print() {
       System.out.println(introduction) ;

       int size = questions.size();
       for (int i = 0 ; i < size ; i++)
          questions.get(i).print(null,i) ;
    }
```

Dans la classe SimpleQuestion, on implémente la méthode print à 2 arguments en appelant simplement la méthode print :

```
public void print(String prefix, int number) {
   print(prefix, number,question);
}
```

Dans la classe ComposedQuestion, on implémente la méthode print à 2 arguments :

```
public void print(String prefix, int number) {
 print(prefix,number,introduction);

 String newPrefix = "" + number ;
 if (prefix != null)
       newPrefix = prefix + "." + newPrefix ;

 int size = sousQuestions.size();
 for (int i = 0 ; i < size ; i++)
       sousQuestions.get(i).print(newPrefix,i) ;
}
```

12. *(3 points)* La variable de classe out de la classe System, celle que l'on utilise pour imprimer dans la console grâce à System.out.print(...) ou System.out.println(...) est de type PrintStream. La classe PrintStream possède un constructeur :

```
public PrintStream(String fileName) throws FileNotFoundException ;
```

En revanche, les méthodes print, println et close ne déclenchent pas d'exception. En cas d'erreur, elles positionnent un drapeau (un *flag*) que l'on peut tester avec la méthode :

```
public boolean checkError() ;
```

Dans la classe ExamSubject, écrire une méthode :

```
public void print(String fileName) ;
```

qui imprime le sujet d'examen dans un fichier de nom fileName. Cette méthode affiche un message d'erreur en cas d'erreur mais ne lève pas d'exception.

(a) On commencera par écrire une méthode `void print(PrintStream out)` qui imprimera dans son argument `out` plutôt que dans `System.out`. On s'inspirera de la réponse à question 11.

(b) On réécrira la méthode `void print()` en utilisant la méthode de (a).

(c) Puis on écrira `void print(String fileName)` qui utilisera également la méthode de (a).

Partout, on importera :
`import java.io.FileNotFoundException`
`import java.io.PrintStream`
ou simplement :
`import java.io.*`
Dans la classe `ExamSubject`, on définit la méthode `print(String fileName)` qui va attraper l'exception `FileNotFoundException`. En cas d'erreur, il n'est pas nécessaire de fermer le `PrintStream` car il n'a pas été ouvert :

```
public final void print(String fileName) {
    PrintStream out = null ;
    try {
        out = new PrintStream(fileName) ;
        print(out) ; // Method to be defined
        if (out.checkError())
            System.err.println(
               "ExamSubject.print(\""+fileName+"\"): unknown error") ;
    } catch (FileNotFoundException e) {
        System.err.println(
           "ExamSubject.print(\""+fileName+"\"): cannot create or open file") ;
    } catch (Exception e) {
        System.err.println(
           "ExamSubject.print(\""+fileName+"\"): " + e.getMessage()) ;
    } finally {
        try { out.close() ; } cach (Exception e) {} ;
    }
}
```

Toujours dans la classe `ExamSubject`, on définit la méthode `print(PrintStream out)` appelé à partir de `print(String fileName)` :

```
public final void print(PrintStream out) {
    out.println(introduction) ;
    int size = questions.size();
    for (int i = 0 ; i < size ; i++)
        questions.get(i).print(out,null,i) ; // Method to be redefined
}
```

Accessoirement dans la classe `ExamSubject`, on peut redéfinir la méthode `print()` :

```
public final void print() {
    print(System.out) ;
}
```

Il faut redéfinir les méthodes de la classe `Question` pour y intégrer le `PrintStream` :

```
import java.io.* ;

public abstract class Question {

    /** Print a question in a stream.
     *  @param out      the stream where to write
     *  @param prefix the prefix to print before the number,
     *                null if there is no prefix
     *  @param number the number of the question
     */
    public abstract void print(PrintStream out, String prefix, int number) ;

    /** Print the number of a question followed by the question in a stream.
     *  @param out      the stream where to write
     *  @param prefix   the prefix to print before the number,
     *                  null if there is no prefix
     *  @param number   the number of the question
     *  @param question the question to be printed
     */
    public final void print(PrintStream out,
                            String      prefix,
                            int         number,
                            String      question) {
        if (prefix != null) {
            out.print(prefix) ;
            out.print('.');
        }
        out.print(number);
        out.print(". ");
        out.println(question);
    }
}
```

Dans la classe `SimpleQuestion`, on implémente le `print` à 3 arguments :

```
public void print(PrintStream out,String prefix, int number) {
    print(out,prefix, number,question);
}
```

De même, dans la classe `ComposedQuestion` :

```
public void print(PrintStream out, String prefix, int number) {
    print(out,prefix,number,introduction);

    String newPrefix = "" + number ;
    if (prefix != null)
        newPrefix = prefix + "." + newPrefix ;

    int size = sousQuestions.size();
    for (int i = 0 ; i < size ; i++)
        sousQuestions.get(i).print(out,newPrefix,i) ;
}
```

FIN DU SUJET