TP Programmation en langage Java

tv <tvaira@free.fr>

Table des matières

Le but de ce TP est de présenter les éléments de base du langage Java, ...

Vous devez rendre une archive compressée contenant les codes sources `FizzBuzz.java`, `TableauCarre.java`, `StringUtils.java`, `TestInverse.java`, `TestPalyndrome.java`, `TestPalyndrome2.java`, `Humain.java`, `TestLuckyLuke`, `Cowboy.java` *et* `Dame.java`.

1 Manipulations

1.1 Séquence n°0 : Avant toute chose ...

Dans un répertoire de travail, créez un répertoire `sequence1` et éditez le fichier `HelloWorld.java` dont le contenu sera le suivant :

```
public class HelloWorld
{
  public static void main (String[] args)
  {
    System.out.println("Hello world!");
  }
}
```

L'éditeur utilisé est quelconque : emacs, vi, geany, gedit, ...Certains éditeurs ont la faculté de colorer la syntaxe et ainsi de donner la possibilité au rédacteur de se rendre compte plus rapidement d'éventuelles erreurs dans l'écriture de mots-clés du langage,...

Cette application java est quasiment la plus simple qu'il soit possible d'écrire. Elle consiste à afficher le message « Hello world ! » sur la console.

1.2 Séquence n°1 : Compilation et exécution

Le but de cette séquence est de se familiariser avec les outils de développement et de documentation Java fournis par Oracle sous le nom de ***Java SE*** *(anciennement Java 2 Standard Edition ou J2SE)* ***SDK*** *(Software Development Kit).*

Comme tout programme source, une application Java nécessite d'être compilée pour pouvoir être exécutée ultérieurement.

Cette opération s'effectue à l'aide de la commande `javac`.

- En vous plaçant dans le répertoire `sequence1`, exécutez la commande `javac` sans paramètre pour obtenir l'aide en ligne.
- Compilez ensuite la classe `HelloWorld` et vérifiez qu'un fichier `.class` a bien été produit dans le même répertoire.
- Vérifiez son type avec la commande `file`.
- Exécutez le programme avec la commande `java`.
- Créez maintenant deux sous-répertoires dans `sequence1` que vous appellerez `src` et `build`. Placez le fichier `HelloWorld.java` dans `src` et, en utilisant l'option `-d` de `javac`, faites en sorte que le fichier `.class` soit produit dans `build`.

Éléments de correction :

- Ligne de commande pour compiler : `javac HelloWorld.java`
- Ligne de commande pour compiler : `file HelloWorld.class`
- Ligne de commande pour exécuter : `java HelloWorld`
- Ligne de commande pour compiler : `javac -d ./build ./src/HelloWorld.java`

Remarque : Il est dans certains cas utile (notamment lorsque l'on transmet l'application via le réseau) de regrouper au sein d'une archive l'ensemble de classes d'une application ou d'une librairie. Une telle archive a en Java un format normalisé et se dénomme un ***jar****. Un outil éponyme*[1] *permet d'archiver des classes.*

1. i.e. du même nom.

1.3 Séquence n°2 : Instructions de base

Cet exercice est extrait du livre « Exemples en Java in a Nutshell ».

Écrire une application Java *FizzBuzz* qui, pour tous les entiers de 1 à 100, affiche sur la console :

- *Fizz* si l'entier est multiple de 5,
- *Buzz* si l'entier est multiple de 7,
- *FizzBuzz* si l'entier est multiple de 5 et de 7,
- la valeur de l'entier sinon.

Éléments de correction :

- Affiche la valeur de l'entier `i` : `System.out.println(Integer.toString(i));`

Cet exercice est extrait du livre « Exercices en Java ».

Écrire un programme *TableauCarre* qui crée et affiche un tableau comportant les valeurs des carrés des n premiers nombres entiers impairs. n sera défini sous forme d'une constante égale à 5.

Éléments de correction :

- Lors de la déclaration d'un tableau, on ne spécifie pas la taille : `char[] tab;` ou `char tab[];` par exemple pour un tableau de `char`
- Le tableau n'existera qu'après l'appel au constructeur : `tab = new char[10];` par exemple pour 10 `char`
- Les constantes se définissent avec `final static`

Un petit `Makefile` pour cette séquence :

```
all: FizzBuzz TableauCarre

FizzBuzz: FizzBuzz.class TableauCarre.class
	@java FizzBuzz

FizzBuzz.class: FizzBuzz.java
	javac FizzBuzz.java

TableauCarre: TableauCarre.class
	@java TableauCarre

TableauCarre.class: TableauCarre.java
	javac TableauCarre.java

clean:
	rm -f *.class *~
```

1.4 Séquence n°3 : Les chaînes de caractères

Le but de cette séquence est de s'exercer à la manipulation de chaînes de caractères.

1.4.1 Inversion de chaîne

Définissez une classe `StringUtils` et ajoutez-y une méthode de signature :
`public static String inverse(String s)` permettant d'inverser les caractères d'une chaîne (i.e. si la chaîne paramètre est "pipo", la méthode retourne "opip").

Écrire une classe de test `TestInverse`.

Éléments de correction :

- Pour accéder à un caractère de la chaîne `s` à la position `pos` : `s.charAt(pos)`
- Autre possibilité en utilisant un `StringBuffer` (ou `StringBuilder`) et sa méthode `reverse()`

1.4.2 Palyndrome

Ajoutez à `StringUtils` une méthode de signature :
`public static boolean estPalyndrome(String s)` permettant de tester si une chaîne de caractères (on ignorera la présence d'espaces en début et fin de chaîne) est un palyndrome (i.e. elle se lit à l'envers comme à l'endroit, comme par exemple "o" ou "tot").

Écrire une classe de test `TestPalyndrome` qui prend la chaîne de caractères à tester sur l'entrée standard.

Éléments de correction :

- Utilisez la méthode `inverse()` créée à l'exercice précédent !
- Pour la saisie, utilisez un `BufferedReader` et sa méthode `readLine()` ou un `Scanner`
- La classe `BufferedReader` nécessite un `InputStreamReader` et le flot d'entrée `System.in`
- Pour les classes `BufferedReader` et `InputStreamReader`, il faut spécifier avec le mot-clé `import` avec le nom complet de chaque classe
- L'utilisation de la méthode `readLine()` peut générer une exception `IOException`, il faut donc soit l'intégrer dans un bloc `try/catch` ou que la méthode `main()` relance les exceptions avec `throws IOException` dans sa définition

Écrire une classe de test `TestPalyndrome2` qui prend la chaîne de caractères à tester sur l'entrée standard jusqu'à la saisie de la valeur entière 0. Ajouter une méthode privée statique `lireInt()` qui retourne la conversion d'un `String` (lu au clavier) en `int` si c'est possible sinon la valeur `-1`.

Éléments de correction :

- Utilisez la méthode `Integer.parseInt()` pour convertir un `String` en `int`
- La méthode `Integer.parseInt()` génère une exception `NumberFormatException` qu'il faudra attraper avec un bloc `try/catch`

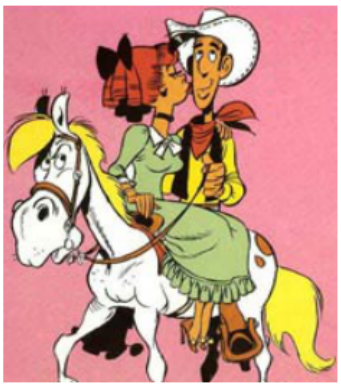
1.5 Séquence n°4 : POO et héritage

Le but de cette séquence est de se familiariser avec la programmation orientée objet en Java.

On désire réaliser un programme *Western* permettant d'écrire facilement des histoires de ... Western.

Dans nos histoires, nous aurons des brigands, des cowboys, des shérifs, des barmen et des dames en détresses ... (à partir d'une idée de Laurent Provot)

Des histoires dans lesquelles un humain arrive, se présente, boit un coup et rencontre une dame coquette :



```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky
Luke et j'aime le coca-cola
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une
jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de coca-cola !
GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```

1.5.1 Notion de classe

Les intervenants de nos histoires sont tous des humains. Notre **humain** est caractérisé par son **nom** et sa **boisson favorite**. La boisson favorite d'un humain est, **par défaut**, de l'eau. Tous les attributs de la classe `Humain` seront privées.

Un humain pourra **parler**. On aura donc une **méthode** `parle(texte)` qui affiche :

`(nom de l'humain) -- texte`

On utilisera cette méthode dès que l'on voudra faire dire quelque chose par un humain.

On veut pouvoir connaître la boisson favorite d'un humain et pouvoir modifier cette dernière. Il faut donc créer deux **méthodes publiques** pour **accéder** à l'**attribut** `boissonFavorite` : la méthode publique `getBoissonFavorite()` est un accesseur (*get*) et `setBoissonFavorite()` est un manipulateur (*set*) de l'attribut `boissonFavorite`.

Un **humain** pourra également **se présenter** (il dit bonjour, son nom, et indique sa boisson favorite), et **boire** (il dira « Ah ! un bon verre de (sa boisson favorite) ! GLOUPS ! »). On veut aussi pouvoir connaître le **nom** d'un **humain** mais il n'est pas possible de le modifier.

Écrire la classe `Humain` afin d'assurer l'exécution du programme de test *TestWestern1* ci-dessous.

```
public class TestWestern1
{
    public static void main (String[] args)
    {
        Humain joe = new Humain("Joe");

        System.out.println("Une histoire sur " + joe.getNom());

        joe.setBoissonFavorite("whisky");

        joe.sePresente();

        joe.boit();
   }
}
```

Écrire la classe de test `TestLuckyLuke` afin d'obtenir une histoire sur *Lucky Luke* :


```
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola

(Lucky Luke) -- Ah ! un bon verre de coca-cola ! GLOUPS !
```

```
Une histoire sur Lucky Luke
(Lucky Luke) -- Bonjour, je suis Lucky Luke et j'aime le coca-cola
(Lucky Luke) -- Ah ! un bon verre de coca-cola ! GLOUPS !
```

1.5.2 Notion d'héritage

Les **dames** et les **cowboys** sont tous des **humains**. Ils ont tous un nom et peuvent tous se présenter. Par contre, il y a certaines différences entre ces **deux classes d'humains**.

Une **dame** est caractérisée par la **couleur de sa robe** (une chaîne de caractères), et par son **état** (libre ou captive).

Elle peut également **changer de robe** (tout en s'écriant « Regardez ma nouvelle robe (couleur de la robe) ! »). On désire aussi changer le mode de présentation des dames car une dame ne pourra s'empêcher de parler de la couleur de sa robe. Et quand on demande son **nom** a une **dame**, elle devra répondre « Miss (son nom) ». Il faut donc **redéfinir les méthodes hérités** `getNom()` et `sePresente()`.

Un **cowboy** est un **humain** qui est caractérisé par sa popularité (0 pour commencer) et un adjectif le caractérisant ("vaillant" par défaut). On désire aussi changer le mode de présentation des cowboys. Un cowboy dira ce que les autres disent de lui (son adjectif).

De même, on veut donner une boisson par défaut à chaque sous-classe d'humain : du lait pour les dames et du whisky pour les cowboys.

Écrire les classes `Cowboy` et `Dame` afin d'assurer l'exécution du programme de test *TestWestern2* ci-dessous.

```
public class TestWestern2
{
    public static void main (String[] args)
    {
        Cowboy lucky = new Cowboy("Lucky Luke");
        Dame jenny = new Dame("Jenny");

        // 1. La rencontre ...
        lucky.sePresente();
        jenny.sePresente();

        // 2. Allons boire un coup ...
        jenny.changeDeRobe("verte");
        lucky.boit();
        jenny.boit();
    }
}
```

Vous devez obtenir :

```
(Lucky Luke) -- Bonjour, je suis le vaillant Lucky Luke et j'aime le whisky
(Jenny) -- Bonjour, je suis Miss Jenny et j'ai une jolie robe blanche
(Jenny) -- Regardez ma nouvelle robe verte !
(Lucky Luke) -- Ah ! un bon verre de whisky ! GLOUPS !
(Jenny) -- Ah ! un bon verre de lait ! GLOUPS !
```

Éléments de correction :

- Pour que la classe `Cowboy` hérite (dérive) de la classe `Humain`, il faut faire : `public class Cowboy extends Humain`
- Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, il doit obligatoirement s'agir de la première instruction du constructeur et ce dernier est désigné par le mot-clé `super`.

Vous pouvez utiliser ce fichier `Makefile` :

```
JAVA         = java
JAVAC        = javac

SRCFILES        := $(wildcard *.java)
#CLASSFILES     := $(wildcard *.class)
CLASSFILES      := $(patsubst %.java, %.class, $(SRCFILES))

.PHONY: clean

all: $(CLASSFILES)

%.class: %.java
        $(JAVAC) $<

histoire-1: $(CLASSFILES)
        $(JAVA) -classpath . TestWestern1

histoire-2: $(CLASSFILES)
        $(JAVA) -classpath . TestWestern2

clean:
        rm -f $(CLASSFILES) *~
```

1.5.3 Notion de paquetage

Un **paquetage** (*package*) est une unité logique renfermant un ensemble de classes ayant un lien (fonctionnel) entre elles. Il est désigné par un nom unique et défini un espace de nommage qui permet

d'éviter les conflits de noms.

On utilise l'instruction `package monpaquet;` en tout début d'un fichier définissant une classe pour indiquer que celle-ci fait partie du `package` nommé `monpaquet`. Attention, toutes les classes compilées faisant partie du paquetage doivent maintenant être présentes dans un répertoire nommé `monpaquet`.

Ajouter l'instruction `package western;` en tout début de fichier pour chaque classe de cette séquence.

Créer l'arborescence suivante : `sequence4/src/western` et `sequence4/build/western` et placez les fichiers sources `*.java` dans `sequence4/src/western`.

Ensuite, on compile et on exécute de la manière suivante dans `./sequence4` :

```
$ javac -d ./build -sourcepath ./src/western -classpath ./build src/western/Cowboy.java
src/western/Dame.java src/western/Humain.java src/western/TestWestern1.java
src/western/TestWestern2.java

$ cd ./build

$ java -classpath ./build western.TestWestern1

$ java -classpath ./build western.TestWestern2
```

L'option `-classpath` (ou la variable d'environnement `CLASSPATH`) contient une liste de chemins conduisant à des répertoires (contenant des classes ou des arborescences de paquetages) et/ou des archives *jar*. Lors de la compilation et de l'exécution, cette variable sert à localiser les fichiers `.class` nécessaires à l'édition de lien.

On peut alors créer le fichier `Makefile.paquetage` dans `./sequence4` :

```
JAVA         = java
JAVAC        = javac

MPACKAGE         = western
MSRCPATH     = ./src/$(MPACKAGE)
MBUILDPATH   = ./build
MCLASSPATH   = $(MBUILDPATH)

SRCFILES        := $(wildcard $(MSRCPATH)/*.java)
CLASSFILES      := $(patsubst $(MSRCPATH)/%.java, $(MBUILDPATH)/$(MPACKAGE)/%.class, $(SRCFILES))

.PHONY: clean

all: $(CLASSFILES)

$(CLASSFILES): $(SRCFILES)
        $(JAVAC) -d $(MBUILDPATH) -sourcepath $(MSRCPATH) -classpath $(MCLASSPATH) $^

histoire-1: $(CLASSFILES)
        (cd $(MBUILDPATH) && $(JAVA) -classpath . $(MPACKAGE).TestWestern1)

histoire-2: $(CLASSFILES)
        (cd $(MBUILDPATH) && $(JAVA) -classpath . $(MPACKAGE).TestWestern2)

clean:
        rm -f $(CLASSFILES) *~
```

1.5.4 Notion d'archive

On peut aussi regrouper l'ensemble des classes dans un fichier d'archive `.jar` :

```
$ cd build

$ jar cvf western.jar western/*.class

$ java western.TestWestern1 -classpath western.jar
```

Remarque : `jar` est un outil distribué avec le JDK de Java qui permet de compresser des classes Java compilées dans une archive.

Il est également possible d'exécuter directement une application contenue dans une archive `jar` avec l'option `-jar` de l'outil `java`. Pour ce faire, il faut cependant avoir ajouté à l'archive un fichier de méta-données que l'on appelle `manifest` indiquant quelle est la classe principale de l'archive.

```
$ cd build

$ vim manifest
Main-Class:  western.TestWestern1

$ jar cvmf manifest western.jar western/*.class

$ java -jar western.jar
```