



Examen de janvier écrit

Durée : **2h00** - Tous les documents *personnels*, manuscrits ou imprimés, sont autorisés.
Tous les appareils électroniques sont interdits et doivent être rangés.

NE RIEN ÉCRIRE AU CRAYON NI À L'ENCRE ROUGE S.V.P.

Ce sujet comporte 5 pages.

Le problème (partie I) est suivi d'un exercice totalement indépendant (II, p. 5), sur les fenêtres et le dessin.

I Problème (ordre d'idée : 14 points) : Expression d'un nombre en différentes langues

Présentation du problème

Les langues naturelles représentent les nombres avec moins de rigueur que la représentation chiffrée. En effet, si certains nombres ont une représentation tout à fait conforme aux principes de numération en base 10, comme 65 ($= 6 \cdot 10 + 5 \cdot 1$), qui se dit

en français de Suisse	soixante cinq	(= soixante + cinq)
en français de France	soixante cinq	(= soixante + cinq)
en hongrois	hatvan öt	(= hatvan + öt)

d'autres, par contre, présentent suivant les langues une variation par rapport à cette normalité logique apparente, comme 72 ($= 7 \cdot 10 + 2 \cdot 1$), qui se dit

en français de Suisse	septante deux	(tout à fait normalement)
en français de France	soixante douze	(au lieu de soixante-dix deux)
en hongrois	hetven kettő	(au lieu de hetven ket)

Les mots du hongrois seront indiqués dans la question E, ceux du français de Suisse dans les questions G et H, et ceux du français de France dans les questions H et G. Vous pouvez déjà vous y reporter.

L'analyse plus détaillée de l'expression en langue naturelle des nombres entiers appartenant $[0,999]$ permet de dire que

- en hongrois, l'expression des nombres se fait en base 10, avec quelques particularités strictement liées à la prononciation
- en français de Suisse, l'expression des nombres se fait en base 10, à l'exception des nombres $\in [10,19]$ pour lesquels on utilise les mots des unités de la base 20 du français de France
- en français de France, l'expression des nombres $\in [0,99]$ se fait majoritairement en base 20, à l'exception des nombres $\in [30,39]$ ou $\in [50,59]$ qui sont en base 10 (par exemple, on dit cinquante trois au lieu de quarante treize); l'expression des centaines se fait en base 10.

Souvent les étrangers se plaignent de la complexité de la numération en français standard (notre dernier cas). On ne peut que les comprendre, car elle mixe allégrement numérations en base 20 et en base 10 !

On s'intéresse à l'expression en hongrois, français de France et français de Suisse d'un nombre entier < 1000 . Le principe général adopté est le suivant : décomposer le nombre en ses trois valeurs de position, en fonction de la base de la langue (centaines, dizaines ou bien vingtaines, unités), puis déterminer

l'expression dans une langue logiquement parfaite (sans tenir compte des particularités), et enfin ajuster cette expression pour la rendre conforme au fonctionnement réel de la langue naturelle. On propose de considérer six classes en héritage, une pour tout ce qui est commun, deux pour la décomposition d'un nombre, trois pour l'expression dans une langue particulière.

La classe *Nombre* comporte tout ce qui est commun aux trois cas : six champs de données (attributs), trois pour les valeurs des trois positions, un pour le mot de la position 2, et deux pour les deux tableaux de mots des positions 1 et 0, et une méthode permettant de déterminer l'expression du nombre dans une langue. L'initialisation des trois champs de position de *Nombre* se fait au niveau des deux classes *Base10* et *Base20Fr* (ceci permet que la valeur de la base soit inconnue de *Nombre*, qui s'avère ainsi indépendante d'une base effective). Les trois classes *ExpressionHongrois*, *ExpressionSuisse* et *ExpressionFrance*, qui héritent de *Base10* ou de *Base20Fr*, comportent chacune une méthode de nom **ajuste** permettant de prendre en compte les particularités de chacune de ces trois langues; elles seront appelées à partir de la classe *Nombre* en définissant dans cette classe une méthode abstraite de même signature (procédé de redéfinition). Une septième classe, *Test* permet de tester le fonctionnement des six classes précédentes.

Très important Toute indentation mal faite ou incorrecte sera notée négativement. Il est de votre responsabilité de *mettre les bons attributs* de champs et de méthodes, *private*, *protected* ou bien *public*.

A dessin de la hiérarchie des classes

Dessinez la hiérarchie de toutes les classes, y compris *Object*.

B classe *Nombre* (début)

Elle comporte les trois positions *pos2*, *pos1* et *pos0*, (dont les valeurs seront initialisées par *Base10* et par *Base20Fr*), le mot de la position 2, *mot2*, et les deux tableaux de mots, *mots1* et *mots0*, pour chacune des positions 1 (dizaines ou vingtaines) et 0 (unités); ces trois valeurs seront initialisées par un constructeur. La méthode **toString** crée une chaîne de caractères correspondant à la valeur de chacune des trois positions présentées sous la forme d'un triplé; exemple (5,8,3). La méthode sans paramètres **dire**, qui renvoie une chaîne de caractère, sera demandée dans une autre question : vous ne mettez ici aucune instruction dans son corps. Enfin une méthode abstraite de nom **ajuste**, à un paramètre chaîne et renvoyant une chaîne, sera redéfinie au niveau des classes correspondant à une langue précise.

Donner la déclaration de la classe *Nombre*, avec la méthode **dire** ayant un corps sans instruction.

```
abstract class Nombre {
    protected int pos2,pos1,pos0;
    private String mot2; private String[] mots1,mots0;

    Nombre() { super(); }
    Nombre(String mt2, String[] mt1, String[] mt0) {
        mot2= mt2; mots1= mt1; mots0= mt0;
    }

    public String dire() {
        ...
        return res;
    }
    abstract protected String ajuste(String s);

    public String toString() {
        return "Les trois positions (" +pos2+", "+pos1+", "+pos0+)";
    }
}
```

C classe *Base10*

Cette classe se compose seulement d'un constructeur. Il permet d'appeler le constructeur de *Nombre* et de décomposer le nombre en ses trois valeurs de position pour une base de numération de 10.

Donner la déclaration complète de la classe *Base10*.

```

abstract class Base10 extends Nombre{

    Base10(int nbr, String mt2, String[] mt1, String[] mt0) {
        super(mt2,mt1,mt0);
        pos2= nbr/(10*10); pos0= nbr%10;
        pos1= (nbr/10)%10;
    }
}

```

D classe *Base20Fr*

Cette classe se compose seulement d'un constructeur. Il permet d'appeler le constructeur de Nombre et de décomposer le nombre en ses trois valeurs de position pour une base de numération de 20, sachant que les centaines sont exprimées en base 10.

Donner la déclaration complète de la classe Base20Fr.

```

abstract class Base20Fr extends Nombre{

    Base20Fr(int nbr, String mt2, String[] mt1, String[] mt0) {
        super(mt2,mt1,mt0);
        pos2= nbr/100; pos0= nbr%20;
        pos1= (nbr%100)/20;
    }
}

```

E classe *ExpressionHongrois*

La numération de cette langue est en base 10. On indique les mots de cette langue grâce à trois champs de données de classe :

```

private final static String[] MOTSPOS0=
    { "" , "egy", "ket", "három", "négy",
      "öt", "hat", "hét", "nyolc", "kilenc" };
private final static String[] MOTSPOS1=
    { "" , "tizen", "huszon", "harminc", "negyven",
      "ötven", "hatvan", "hetven", "nyolcvan", "kilencven" };
private final static String MOTPOS2 = "szaz";

```

On redéfinit ici la méthode `ajuste` de Nombre. Les ajustements de cette langue sont les suivants : remplacer ket par kettő, tizen par tiz et huszon par husz lorsque l'expression sans ajustement se termine respectivement par ket, tizen ou huszon. Attention à ne pas oublier le cas du nombre 0, qui se dit nulla en hongrois. Consultez les définitions de méthodes de la classe String données en fin de texte.

Donner la déclaration complète de la classe ExpressionHongrois, en ne répétant pas la définition des trois champs de données ci-dessus (indiquez : // les trois champs de données).

```

class ExpressionHongrois extends Base10{
    private final static String[] MOTSPOS0= { "" , "egy", "ket", "három", "négy",
                                              "öt", "hat", "hét", "nyolc", "kilenc" };
    private final static String[] MOTSPOS1={ "" , "tizen", "huszon", "harminc", "negyven",
                                              "ötven", "hatvan", "hetven", "nyolcvan", "kilencven"
};
    private final static String MOTPOS2 = "szaz";

    ExpressionHongrois(int nombre) {
        super(nombre, MOTPOS2, MOTSPOS1, MOTSPOS0);
    }

    protected String ajuste(String s) {
        String res= s;
        if (res.endsWith("ket")) res= res+"tő";
        else if (res.endsWith("tizen")) res= res.substring(0, res.length()-2);
            else if (res.endsWith("huszon")) res= res.substring(0, res.length()-2);
        if (res.equals("")) res= "nulla";
        return res;
    }
}

```

```
}  
}
```

F classe *Nombre* : méthode *dire*

La méthode *dire* de *Nombre* compose une chaîne de caractères correspondant à une expression en langue standard, avant les ajustements particuliers nécessités par la langue. Elle est donc sans paramètre et renvoie une chaîne.

Donner la déclaration de la méthode *dire* de la classe *Nombre*. Commencez à construire votre chaîne résultat plutôt à partir de la position 0. Faites attention à traiter le problème des espacements entre mots, un seul espacement entre deux mots successifs; ne pas commencer ni terminer la chaîne par un espace.

```
abstract class Nombre {  
    protected int pos2, pos1, pos0;  
    private String mot2; private String[] mots1, mots0;  
  
    Nombre() { super(); }  
    Nombre(String mt2, String[] mt1, String[] mt0) {  
        mot2= mt2; mots1= mt1; mots0= mt0;  
    }  
  
    public String dire() {  
        String res, posStr;  
        res= mots0[pos0];  
        posStr= mots1[pos1];  
        res= posStr+(posStr.length()!=0 && res.length()!=0?" ":"")+res;  
        posStr= (pos2>=2?mots0[pos2]+" ":"")+ (pos2!=0?mot2:"");  
        res= posStr+(posStr.length()!=0 && res.length()!=0?" ":"")+res;  
        res= ajuste(res);  
        return res;  
    }  
    abstract protected String ajuste(String s);  
  
    public String toString() {  
        return "Les trois positions (" +pos2+", "+pos1+", "+pos0+)";  
    }  
}
```

G classe *ExpressionSuisse*

La numération de cette langue est en base 10. Les mots de la langue pour la numération sont les suivants :

(un,deux,trois,quatre,cinq,six,sept,huit,neuf)
(dix,vingt,trente,quarante,cinquante,soixante,septante,huitante,nonante)
cent

L'ajustement concerne ici les nombres $\in [10,19]$ qui utilisent la deuxième partie des symboles de la position 0 de la base 20 du français de France. Ces symboles sont récupérés grâce à un accesseur de classe de la classe *ExpressionFrançais*. Attention à ne pas oublier le cas du nombre 0 (zéro). Consultez les définitions de méthodes de la classe *String* données en fin de texte.

Donner la déclaration complète de la classe *ExpressionSuisse*.

```
class ExpressionSuisse extends Base10{  
    private final static String[] MOTSPOS0= {"", "un", "deux", "trois", "quatre",  
                                             "cinq", "six", "sept", "huit", "neuf" };  
    private final static String[] MOTSPOS1=  
        { "", "dix", "vingt", "trente", "quarante",  
          "cinquante", "soixante", "septante", "huitante", "nonante" };  
    private final static String MOTPOS2 = "cent";  
  
    private final String[] motsDizaine;  
  
    ExpressionSuisse(int nombre) {  
        super(nombre, MOTPOS2, MOTSPOS1, MOTSPOS0);  
    }  
}
```

```

    motsDizaine= ExpressionFrancais.getMotsPos0();
}

public String ajuste(String s) {
    String res= s;
    int index= s.indexOf("dix ");
    if (index!=-1) { res= s.substring(0,index)+motsDizaine[pos0+10]; }
    if (res.equals("")) res= "zéro";
    return res;
}
}

```

H classe *ExpressionFrancais*

La numération de cette langue est majoritairement en base 20 pour [0,99] et en partie en base 10 pour [30,39] et [50,59] et pour les centaines. Les mots de la langue pour la numération sont les suivants :

(un,deux,trois,quatre,cinq,six,sept,huit,neuf,dix,onze,douze,treize,quatorze,quinze,seize,dix-sept,dix-huit,dix-neuf)
(vingt,quarante,soixante,quatre-vingt)
cent

L'ajustement concerne la deuxième moitié des deuxièmes et troisièmes vingtaines, où on laisse la base vingt au profit de la base 10, ces deux dizaines correspondant exactement à celles du français de Suisse. Pour simplifier, on ne tiendra pas compte des « et » éventuellement insérées entre les mots des positions 1 et 0 lorsque la position 0 correspond à un ou onze (exemples trente et un, soixante et onze). Utilisez les valeurs numériques des positions 1 et 0. Attention à ne pas oublier le cas du nombre 0.

Donner la déclaration complète de la classe ExpressionFrancais.

```

class ExpressionFrancais extends Base20Fr{
    private final static String[] MOTSPOS0= {"", "un", "deux", "trois", "quatre",
                                             "cinq", "six", "sept", "huit", "neuf",
                                             "dix", "onze", "douze", "treize",
                                             "quatorze",
                                             "quinze", "seize", "dix-sept", "dix-huit", "dix-neuf"};
    private final static String[] MOTSPOS1= {"", "vingt", "quarante", "soixante", "quatre-vingt"};
};
private final static String MOTPOS2 = "cent";
public static String[] getMotsPos0() { return MOTSPOS0; }

ExpressionFrancais(int nombre) {
    super(nombre, MOTPOS2, MOTSPOS1, MOTSPOS0);
}

protected String ajuste(String s) {
    String res= s;
    if ( (pos1==1 || pos1==2) && pos0>9 )
        res= new ExpressionSuisse(pos2*100+pos1*20+pos0).dire();
    if (res.equals("")) res= "zéro";
    return res;
}
}

```

I classe *Test*

Déclarer un tableau à trois cases de Nombre. Pour chacune des valeurs entières de 0 à 215, initialiser les trois cases du tableau avec des objets pour successivement le hongrois, le français de Suisse et le français de France et, à l'aide d'une boucle, afficher sur une même ligne ces trois expressions en langue naturelle pour la valeur entière considérée, et passer alors à la ligne.

L'affichage obtenu est du type suivant :

```

...
szaz nyolcvan kilenc - cent huitante neuf - cent quatre-vingt neuf -
szaz kilencven - cent nonante - cent quatre-vingt dix -
...

```

Donner la déclaration complète de la classe *Test*.

```

class Test {
    public static void main(String[] a) {
        Nombre[] nbr= new Nombre[3];
        for (int i=0; i<215; i++) {
            nbr[0]= new ExpressionHongrois(i);
            nbr[1]= new ExpressionSuisse(i);
            nbr[2]= new ExpressionFrancais(i);
            for (int l=0; l<nbr.length; l++) System.out.print(nbr[l].dire()+" - ");
            System.out.println();
        }
    }
}

```

Quelques méthodes de la classe *String*

public boolean **endsWith**([String](#) suffix)

(endsWith signifie finit par)

Tests if this string ends with the specified suffix.

Returns: true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise.

public boolean **equals**([Object](#) anObject)

(equals signifie est égal à)

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Returns: true if the String are equal; false otherwise.

public int **indexOf**(int ch)

(indexOf signifie indice de)

Returns: the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

public int **indexOf**(int ch,int fromIndex)

(indexOf signifie indice de)

Returns: the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

There is no restriction on the value of fromIndex. If it is negative, it has the same effect as if it were zero: this entire string may be searched. If it is greater than the length of this string, it has the same effect as if it were equal to the length of this string: -1 is returned.

public int **indexOf**([String](#) str) (indexOf signifie indice de)

Returns: if the string argument occurs as a substring within this object, then the index of the first character of the first such substring is returned; if it does not occur as a substring, -1 is returned.

public int **indexOf**([String](#) str,int fromIndex) (indexOf signifie indice de)

Returns: the index within this string of the first occurrence of the specified substring, starting at the specified index; if it does not occur as a substring, -1 is returned.

public int **length**() (length signifie longueur)

Returns: the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

public boolean **startsWith**([String](#) prefix) (startsWith signifie commence par)

Tests if this string starts with the specified prefix.

Returns: true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise. Note also that true will be returned if the argument is an empty string.

public boolean **startsWith**([String](#) prefix,int toffset) (startsWith signifie commence par)

Tests if this string starts with the specified prefix beginning a specified index.

Returns: true if the character sequence represented by the argument is a prefix of the substring of this object starting at index toffset; false otherwise. The result is false if toffset is negative or greater than the length of this String object.

public [String](#) **substring**(int beginIndex) (substring signifie sous-chaîne)

Returns: a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

public [String](#) **substring**(int beginIndex,int endIndex) (substring signifie sous-chaîne)

Returns: a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

II Exercice indépendant (ordre d'idée : 6 points)

On considère un jeu éducatif, de type jeu de parcours, comportant des cartes (à jouer). Certaines d'entre elles représentent des feux tricolores, soit au rouge, soit au vert (autorisant ou non le passage). Pour dessiner ces cartes à l'écran, on généralise un tout petit peu en disant qu'il convient de pouvoir afficher des cartes « feux tricolores » avec un seul des trois feux allumé, soit le vert, soit l'orange, soit le rouge.

Une carte correspond à un composant de fenêtre. Le fond de ce composant est noir, et les trois feux sont représentés par trois disques, à leur place habituelle. Si un feu est éteint, il est en blanc, s'il est allumé il est de sa couleur, soit vert orangé ou rouge du bas vers le haut. La classe *Un Feu* décrit ce composant de fenêtre. Chacune des possibilités pour des feux tricolores, feu au vert, à l'orange ou au rouge correspond à une instantiation particulière de cette classe.

La classe *UnFeu* comporte trois champs de données

```
private Color vrt, orng, rge;
```

La valeur de ces champs détermine la couleur d'affichage des trois disques représentant le feu tricolore. La valeur de ces couleurs est initialisée par le constructeur. Il comporte cinq paramètres

```
UnFeu(int larg, int haut, boolean vert, boolean orange, boolean rouge)
```

larg et **haut** correspondent à la dimension du composant; **vert** est à true si **vrt** doit être initialisé à green, **orange** est à true si ...

Chaque disque occupe en hauteur 85% du tiers de la hauteur de la carte.

C'est dans une fenêtre placée en (100,100) et de dimension (350,260) que seront placés les cartes. La classe *TestExo* placera trois cartes de taille (90,220) dans la fenêtre, avec des paramètres effectifs true,false,false puis false,true,false puis false,false,true correspondant respectivement aux trois cas feu au vert, feu à l'orange et feu au rouge.

```
import javax.swing.*;
import java.awt.*;

public class UneFenetre extends JFrame{

    public UneFenetre(String titre) {
        super(titre);
        setBounds(100,100,350,260);
        getContentPane().setLayout(new FlowLayout());
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}

*****

import javax.swing.*;
import java.awt.*;

class UnFeu extends JPanel{
    private Color vrt, orng, rge;

    UnFeu(int larg, int haut, boolean vert, boolean orange, boolean rouge) {
        setPreferredSize(new Dimension(larg, haut));
        setBackground(Color.black);
        if (vert) vrt= Color.green; else vrt= Color.white;
        if (orange) orng= Color.orange; else orng= Color.white;
        if (rouge) rge= Color.red; else rge= Color.white;
    }

    private void tracer(Graphics h) {
        double decale= getHeight()/3, diametre= decale*0.85;
        int deb= (int)(decale*0.075), diam= (int)diametre, gau= (int)(getWidth()/2-
diametre/2);
```

```

        h.setColor(rge) ; h.fillOval(gau,deb          ,diam,diam);
        h.setColor(orange); h.fillOval(gau,deb+(int)decale    ,diam,diam);
        h.setColor(vrt) ; h.fillOval(gau,deb+(int)(2*decale),diam,diam);
    }
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        tracer(g);
    }
}

*****

class TestExo {

    public static void main (String[] a) {
        UneFenetre support= new UneFenetre("trois cartes \"feux tricolores\"");
        support.getContentPane().add(new UneFeu(90,220,true,false,false));
        support.getContentPane().add(new UneFeu(90,220,false,true,false));
        support.getContentPane().add(new UneFeu(90,220,false,false,true));
        support.setVisible(true);
    }

}

```