Interrogation d'informatique
PCC-ASINSA 2ème année - Janvier 2017

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Durée totale : *3h*
Documents autorisés : *Toutes notes personnelles ou du cours*
Attention : les téléphones portables sont interdits.

- Le barème est indicatif et le sujet est sur 19 pages.
- Les exercices sont indépendants.
- Un programme **mal indenté**, **mal commenté** ou avec de **mauvais choix de noms de variables** sera **sanctionné** *(jusqu'à -1 point)*.

À méditer avant de commencer :
« Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. »*Martin Golding*

Barème indicatif :
- Partie 1 : 9 points
- Partie 2 : 7 points
- Partie 3 : 4 points

Dernières précisions importantes à lire attentivement :
- les 3 parties sont indépendantes. Vous pouvez les traiter dans l'ordre que vous voulez.
- pour chaque question, vous devrez utiliser (dans la mesure du possible) les méthodes présentées dans les questions précédentes, même si vous ne les avez pas codées ;
- l'efficacité des algorithmes proposés fera l'objet d'une attention particulière ;
- sauf indications contraires, la visibilité des attributs des objets créés devra être `public` ;
- la longueur du sujet est essentiellement due à la longueur des explications mais ne reflète pas nécessairement la longueur de vos réponses, donc pas d'inquiétude !
- le code java donné dans vos réponses devra être le plus concis possible et donc exploiter le plus possible les structures d'héritage. Par exemple, dans une classe fille on pourra faire appel à `super.maMethode()` pour utiliser la méthode `maMethode()` définie dans la classe mère.

1 Gestion d'un parcours sur une autoroute à péage

En tirant profit de la Programmation Orientée Objet (POO), on désire modéliser de façon très simplifié le parcours d'un véhicule sur une autoroute payante. Il est à noter que le problème est relativement ouvert et plusieurs solutions sont possibles.

Le parcours se fera toujours dans un sens unique. Une autoroute est composée d'une suite de péages dans la limite de 50 péages maximum. Chaque péage est caractérisé par un nom et sa position kilométrique sur l'autoroute par rapport au péage précédent.

Un véhicule, identifié par sa plaque d'immatriculation, voyageant sur une autoroute traverse les péages les uns après les autres. Ainsi, une fois le premier péage passé, il traverse un ou plusieurs péages de transit et lorsqu'il termine son parcours, il franchit le péage de sortie où il paye un prix qui dépend de la distance parcourue et de son type.

Il existe différents types de véhicules : des motos, des voitures et des camions. Quel que soit le véhicule, le prix correspondant à la traversée d'un péage correspond aux éléments suivants :

- tous les véhicules sont caractérisés par un coefficient kilométrique. À chaque fois qu'un péage est traversé on calcule un prix en multipliant ce coefficient par la distance parcourue sur l'autoroute. Ce coefficient est de 0.05€/km pour une moto, 0.1€/km pour une voiture et 0.15€/km pour un camion.
- tous les véhicules doivent pouvoir calculer une majoration fixe qui est ajoutée au prix kilométrique lorsque le dernier péage (le péage de sortie) est traversé. Cette majoration est nulle pour une moto. Elle est donnée par le nombre de passagers multiplié par 10 pour une voiture et par le nombre de tonnes de marchandises multiplié par 0.5 pour un camion.

Un véhicule qui effectue un parcours sur autoroute traverse donc successivement les péages de cette autoroute. Pour tout véhicule effectuant un parcours, on veut pouvoir à tout moment :

- afficher toutes les données relatives au véhicule (*cf.* figure 1) ;
- connaître le nom du dernier péage traversé ;
- les frais de péage (c'est à dire le prix correspondant au dernier péage traversé). Ce cumul devra donc être mis à jour à chaque fois que le véhicule traverse un péage.

On vous donne ci-dessous une partie de la déclaration de la classe `Parcours`, le programme principal et le résultat de son exécution.

La classe `Parcours` :

```
1  public class Parcours {
2
3    // Attributs à décrire (si nécessaire)
4
5    public Parcours(String nom) {
6      // Code à écrire
7    }
8    public String toString(){
9      // Code à écrire
10   }
11   public void ajouterPeage(Peage p) {
12     // Code à écrire
13   }
14   public void faireUnVoyage(Vehicule v) {
15     // Code à écrire : Appel de la méthode traverser(Vehicule a) de chacun des péages
           successifs
16   }
17 }
```

La classe principale pour effectuer un parcours de Paris à Marseille en passant par Lyon :

```java
public static void main(String[] args){
    Parcours versLeSoleil = new Parcours("versLeSoleil");
    versLeSoleil.ajouterPeage(new Peage("Paris"));
    versLeSoleil.ajouterPeage(new PeageTransit("Lyon", 600));
    versLeSoleil.ajouterPeage(new PeageSortie("Marseille", 400));
    versLeSoleil.faireUnVoyage(new Moto ("1234AB69") );
    versLeSoleil.faireUnVoyage(new Voiture ("5678CD69", 5) );
    versLeSoleil.faireUnVoyage(new Camion ("9012EF69", 200) );
}
```

La figure 1 est une capture d'écran du terminal suite à l'exécution de la classe principale :

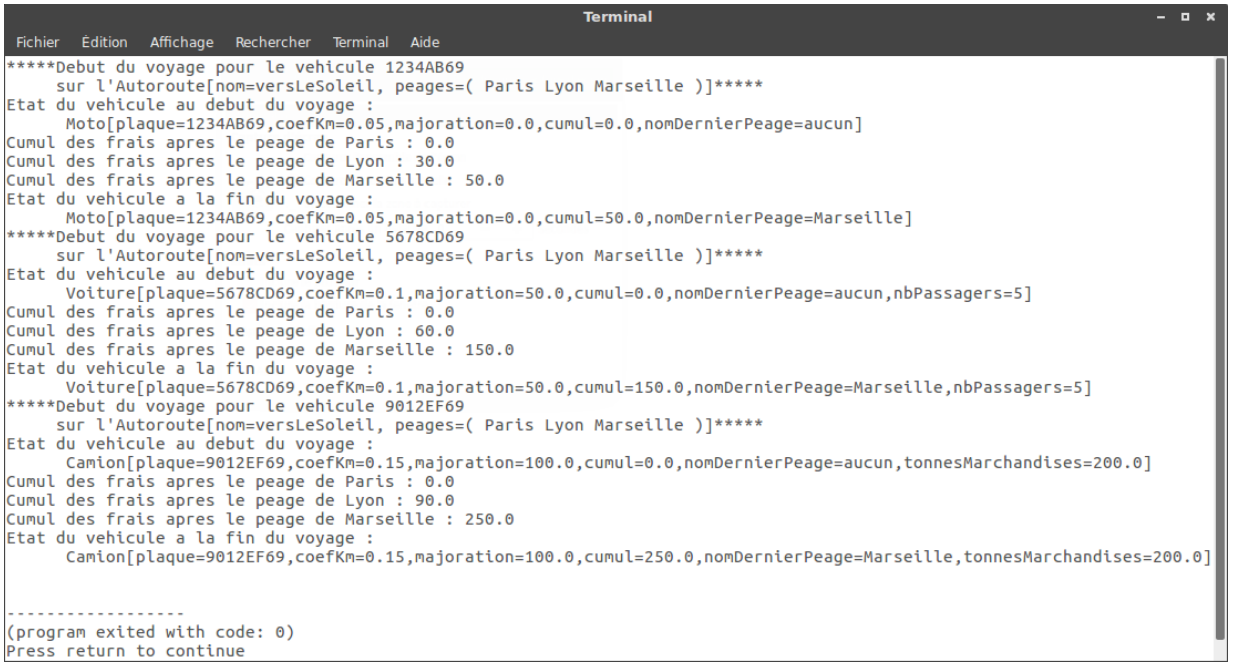


FIGURE 1 – Affichage suite à l'exécution de la classe principale

(Q1.1) Modélisation

On donne ci-dessous le diagramme de la hiérarchie des classes modélisant les péages (figure 2). La première ligne est le nom de la classe, la seconde ses attributs et la troisième ses méthodes. La classe mère est la classe `Peage` et les deux autres des classes filles. Donner le diagramme des classes modélisant les véhicules en utilisant les mêmes conventions d'écriture que pour le diagramme des péages. On impose que toutes les classes de cette hiérarchie comportent au moins les méthodes `toString()` et `calculMajoration()`.

Voici une proposition de diagramme UML pour `Vehicule` *et ses descendants :*
Tous les attributs de `Vehicule`
Les méthodes calculMajoration() et toString() dans chaque classe qui doivent être redéfinies
Ne pas pénaliser si les classes et les méthodes abstraites n'apparaissent pas ici. Cela se verra dans le code.
Les 3 classes héritières de `Vehicule`
Les attributs spécifiques de `Voiture` *et de* `Camion`

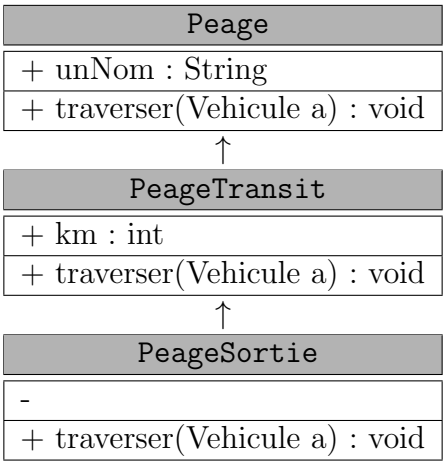
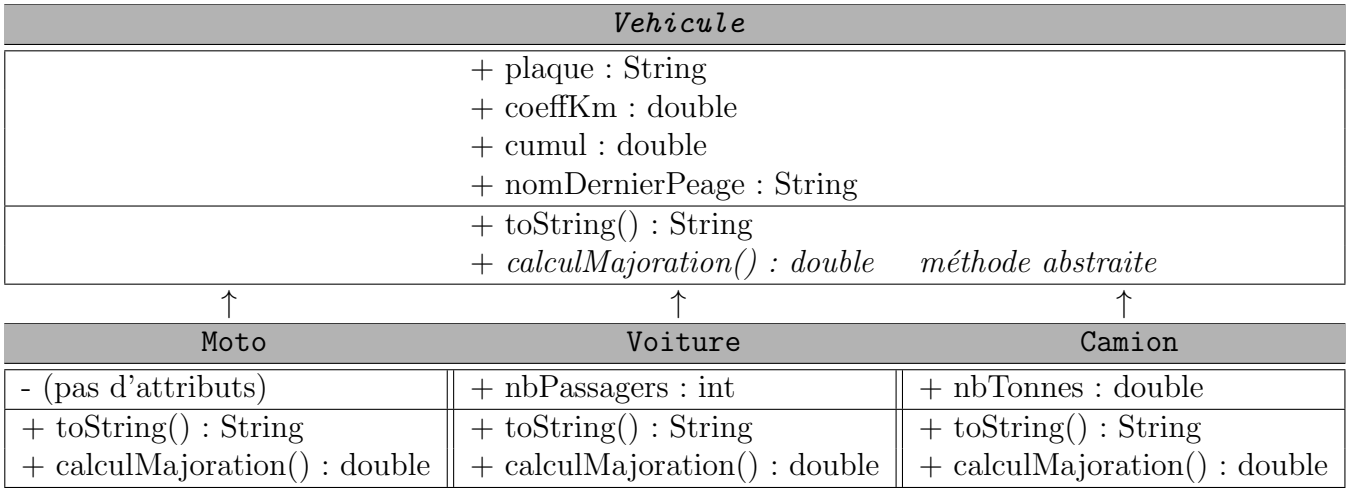


FIGURE 2 – Diagrammes UML des classes `Peage, PeageTransit et PeageSortie`



(Q1.2) Programmation des véhicules

Donnez tout le code java nécessaire à la modélisation des véhicules.

Pour la classe `Vehicule` *:*

```
public abstract class Vehicule{

    public String plaque;
    public double coefKm;
    public double cumul;
    public String nomDernierPeage;

    public Vehicule( String unePlaque ){
        plaque = unePlaque;
        nomDernierPeage="aucun";
        cumul = 0.0;
    }

    public abstract double calculMajoration();

    public String toString(){
        return  "plaque="+plaque+",coefKm="+coefKm+
                ",majoration="+this.calculMajoration()+",cumul="+cumul+
                ",nomDernierPeage="+nomDernierPeage;
    }

}
```

Les attributs (à adapter selon le diagramme UML définie précédemment)
Le constructeur (qui peut contenir unCoeffKm éventuellement)
Initialisation des attributs dans le constructeur et non pas lors de leur déclaration
La méthode calculMajoration() qui est abstraite et donc la classe est aussi abstraite

Pour la classe `Moto` *:*

```
public class Moto extends Vehicule{

    public Moto( String unePlaque ){
        super( unePlaque );
        coefKm = 0.05;
    }

    public double calculMajoration(){
        return (0.0);
    }

    public String toString(){
        return "      Moto["+super.toString()+"]";
    }
}
```

Le constructeur avec appel à `super`
Initialisation des attributs dans le constructeur
Compléments de la méthode `toString()` *avec appel à super*
La méthode calculMajoration() qui doit être redéfinie

Pour la classe `Voiture` *:*

```
public class Voiture extends Vehicule{

    public int nbPassagers;

    public Voiture( String unePlaque, int nb ){
        super( unePlaque );
        nbPassagers = nb;
        coefKm = 0.1;
    }

    public double calculMajoration(){
        return (10.0*nbPassagers);
    }

    public String toString(){
        return  "       Voiture["+super.toString()+
                ",nbPassagers="+nbPassagers+"]";
    }
}
```

Le constructeur avec appel à `super`
Initialisation des attributs dans le constructeur
Compléments de la méthode `toString()` *avec appel à super*
Ajout de l'attribut `nbPassagers`
La méthode calculMajoration() qui doit être redéfinie
Pour la classe `Camion` *:*

```
public class Camion extends Vehicule{

    public double tonnesMarchandises;

    public Camion( String unePlaque, double Poids ){
        super( unePlaque );
        tonnesMarchandises = Poids;
        coefKm = 0.15;
    }

    public double calculMajoration(){
        return (0.5*tonnesMarchandises);
    }

    public String toString(){
        return  "       Camion["+super.toString()+
                ",tonnesMarchandises="+tonnesMarchandises+"]";
    }
}
```

Le constructeur avec appel à `super`
Initialisation des attributs dans le constructeur
Compléments de la méthode `toString()` *avec appel à super*
Ajout de l'attribut `nbTonnesMarchandises`
La méthode calculMajoration() qui doit être redéfinie

(Q1.3) En route !

Donnez le code java de la méthode `traverser(Vehicule a)` des différents péages. Ces méthodes ne feront aucun affichage dans le terminal.

Pour la classe `Peage` *:*

```
public void traverser(Vehicule a){
    a.nomDernierPeage = nom;
    a.cumul = 0.0;
}
```

Mise à jour du nom du dernier péage traversé
initialisation de cumul à 0

Pour la classe `PeageTransit` *:*

```
public void traverser(Vehicule a){
    a.nomDernierPeage = nom;
    a.cumul += a.coefKm*km;
}
```

Mise à jour du nom du dernier péage traversé (ou appel à super.traverser(a))
Incrémentation de la valeur de cumul selon le coeff et le nombre de km parcourus

Pour la classe `PeageSortie` *:*

```
public void traverser(Vehicule a){
    super.traverser(a);
    a.cumul += a.calculMajoration();
}
```

Appel à super.traverser(a) (ou mise à jour du nom du dernier péage et du cumul)
Ajout de la majoration à cumul

(Q1.4) Gestion du parcours sur l'autoroute

Donnez le code java complet de la classe `Parcours` à partir du code donné au début de l'énoncé (section 1).

Pour la classe `Parcours` *:*

```
public class Parcours {

    public String nom;
    public Peage[] tabPeage;
    public int nbPeage;

    public Parcours(String unNom) {
        nom = unNom;
        tabPeage = new Peage[50];
        nbPeage = 0;
    }

    public void ajouterPeage(Peage p) {
        tabPeage[nbPeage] = p;
        nbPeage++;
    }

    public String toString() {
        String s = "Autoroute[nom=" + nom + ", peages=( ";

        for (int i=0; i<nbPeage; i++)
            s = s + tabPeage[i].nom + " ";

        s = s +")]";
        return s;
    }

    public void faireUnVoyage(Vehicule vehicule) {
        System.out.println( "*****Debut du voyage pour le vehicule "+
                            vehicule.plaque);
        System.out.println("      sur l'"+this.toString()+ "*****");
        System.out.println("Etat du vehicule au debut du voyage :" );
        System.out.println(vehicule);

        for (int i=0; i<nbPeage; i++){
            tabPeage[i].traverser(vehicule);
            System.out.println("Cumul des frais apres le peage de "+
                               tabPeage[i].nom+" : "+vehicule.cumul);
        }
        System.out.println("Etat du vehicule a la fin du voyage :" );
        System.out.println(vehicule);
    }
}
```

déclaration des attributs
Constructeur et initialisation des attributs dans le constructeur
méthodes ajouterPeage et toString
méthode faireUnVoyage

2 Puissance 4

Dans cette section, on cherchera à programmer un jeu inspiré du jeu Puissance 4™ dont la première édition date de 1974 (figure 3).
Règles de base du Puissance 4 :
Il s'agit d'un jeu à deux joueurs. À tour de rôle, chacun des deux joueurs joue un pion de sa couleur dans une colonne. Le pion tombe dans la colonne jusqu'à atteindre la position vide la plus basse de la colonne. Le premier qui aligne 4 de ses pions (verticalement, horizontalement ou en diagonale) gagne la partie.

FIGURE 3 – Le jeu Puissance 4 dans sa première édition

Dans cette section, nous nous intéresserons à la programmation de ce jeu. Le jeu se fera sur une grille carrée et pourra se jouer à un joueur, l'autre joueur étant géré par le programme. La figure 4 représente le plateau de jeu sur lequel on va jouer. Les cases grisées servent à connaître les coordonnées d'un pion. Pour tout l'exercice on considérera que le premier indice du tableau correspond aux colonnes et le second indice aux lignes. La case en haut à gauche est d'indice [0,0]. Ainsi si un joueur place un pion dans la colonne 4, son pion arrivera en position [4,6]. Si, lors d'un coup suivant, un autre pion est placé dans la même colonne, ce dernier arrivera en position [4,5] et ainsi de suite. Dès qu'une colonne est pleine, il n'est plus possible de joueur dans cette colonne. Le match est déclaré comme nul lorsque la grille est pleine et qu'aucun joueur n'a réussi à aligner 4 de ses pions.

(Q2.1) La grille de jeu

Par souci de simplification, on considérera la grille de jeu comme un tableau 2D d'entier dont la taille sera, par défaut, égale à 7 mais pourra être changée lors du début de la partie.
Pour initialiser la grille, on utilisera la méthode ci-dessous qui se trouvera dans la classe `Grille`. Cette méthode permet d'affecter la valeur 0 à chaque case pour symboliser que celle-ci est vide.

	0	1	2	3	4	5	6
0	.	.	.	.	.	.	.
1	.	.	.	.	.	.	.
2	.	.	.	.	.	.	.
3	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.
6	.	.	.	.	.	.	.

FIGURE 4 – Exemple d'un plateau de jeu

Par la suite, le pion du joueur sera symbolisé par le numéro du joueur (1 = rouge, 2 = jaune).

```
1  /**
2   * Pour initialiser la grille de début de jeu
3   * @param taille la taille de l'aire de jeu
4   */
5  public void initGrille(int taille){
6      this.taille=taille;
7      tab = new int[taille][taille];
8      for (int i=0;i<taille;i++)
9          for (int j=0;j<taille;j++)
10             tab[i][j]=0;
11 }
```

(Q2.1).1 Attribut(s) et constructeur(s) de la grille

D'après le code ci-dessus, définissez les attributs de la classe `Grille`.
Proposez le (ou les) constructeur(s) adéquat(s) permettant de créer une grille dont la taille est définie par l'utilisateur (*via* le programme principal) ou égale à 7 par défaut (*i.e.* si l'appel au constructeur se fait sans paramètre, on imposera une taille égale à 7). Attention, lors de l'appel au constructeur avec un paramètre inférieur à 4 (*i.e.* une grille de moins de 4 colonnes), on imposera une taille du plateau égale à 4.

D'après le code, on a besoin de 2 attributs :
`public int[][] tab;`
`public int taille;`
D'après les consignes, ces attributs doivent être `public` *mais on peut accepter* `private` *ou* `protected`*. Par contre il faut respecter les noms et les types.*
Concernant les constructeurs, il en faut 2 :

```
/**
 * Le constructeur sans parametre (taille par default = 7)
 */
public Grille(){
    initGrille(7);
}


/**
 * Le constructeur
 * @param : la taille de l'aire de jeu, si taille < 4, on impose
 * une taille a 4
 */
public Grille(int laTaille){
    if (laTaille<4)
        initGrille(4);
    else
        initGrille(laTaille);
}
```

Chaque constructeur doit faire appel à la méthode `initGrille`.
Dans le second constructeur, il faut imposer une taille égale à 4 si le paramètre du constructeur est inférieur à 4.

(Q2.1).2 Désolé, on est complet ici, allez voir à côté !

Lorsqu'une colonne est remplie de pions, on ne peut plus jouer dans cette colonne. Proposer le code de la méthode `colonnePleine` dont la signature est :

```
1   /**
2    * Vérifie si la colonne est pleine
3    * @param numCol le numéro de la colonne concernée
4    * @return true si la colonne est pleine
5    */
6   public boolean colonnePleine(int numCol){
7     // Code à compléter ici
8   }
```

Voici le code correspondant :

```
public boolean colonnePleine(int numCol){
    return (tab[numCol][0] != 0);
}
```

Il faut simplement tester que la ligne la plus haute soit vide ou pas
Si le code contient une boucle for ou while, seulement la moitié des points

(Q2.1).3 Jeu, Set et Match ... nul, veuillez réessayer

Lorsque la grille est pleine de pions sans qu'aucun des joueurs n'ait réussi à aligner 4 de ses pions, la partie est déclarée nulle. Après chaque coup joué, il est donc nécessaire de vérifier que la grille n'est pas pleine. Proposer le code de la méthode `estPlein` dont la signature est :

```
1   /**
2    * Vérifie s'il est encore possible de placer des pions
3    * @return true si le tableau est plein
4    */
5   public boolean estPlein() {
6    // Code à insérer ici
7   }
```

NB : *U*tiliser la méthode `colonnePleine` de la question précédente même si vous ne l'avez pas codée.

Voici le code attendu (utilisation d'une boucle while et de la méthode précédente) :

```
public boolean estPlein() {
    boolean complet=true;
    int col=0;
    while (complet && col<taille){
        if (!colonnePleine(col)){
            complet=false;
        }
        col++;
    }
    return complet;
}
```

Parcourir juste les colonnes et faire appel à la méthode `colonnePleine`
Utilisation d'une boucle while
Renvoie d'un booléen correspondant à l'énoncé

ou autre solution en comptant le nombre de colonne pleine

```
public boolean estPlein() {
    int col =0;
    while ( col < taille && !colonnePleine (col)) // conditions dans cet ordre uniquement
        col ++;
    return (col == taille);
}
```

ou autre solution en utilisant une boucle for (ce qui est moins bien) ou une double boucle for (qui est encore moins bien)

```
public boolean estPlein() {
    boolean complet=true;
    // On cherche une case vide sur la ligne la plus haute, i.e. une case contenant 0.
    for (int col = 0; col < taille; col++) {
        if (!colonnePleine(col)) {
            complet= false;
        }
    }
    return complet;
}
```

Parcourir simplement la ligne la plus haute (d'indice 0) et chercher une case vide
Renvoie d'un booléen correspondant à l'énoncé
Si il y a une double boucle for, il faut pénaliser

(Q2.1).4 Placer un pion dans une colonne de la grille

On supposera que le programme demande au joueur la colonne dans laquelle il souhaite jouer. La méthode `joueCoup` vérifie que le coup est réalisable et place le pion dans la grille. Le numéro de la colonne où le joueur souhaite jouer est passé en paramètre. On rappelle que le pion sera placé sur la ligne la plus basse (et donc à l'indice le plus élevé) de la grille (*cf.* figure 4). La méthode `joueCoup` renverra la valeur `false` si le coup n'est pas possible (*i.e.* si la colonne est pleine) et la valeur `true` dans le cas contraire.
Compléter le code de cette méthode dont la signature est :

```java
/**
 * Pour tester et placer un pion d'un joueur
 * @param col le numéro de la colonne jouée
 * @param joueur le joueur concerné (représenté par un entier)
 * @return true si le coup a été validé
 */
public boolean joueCoup(int col, int joueur){
 // Code à insérer ici
}
```

Voici le code correspondant :

```java
public boolean joueCoup(int col, int joueur) {
    boolean valide = false;
    if ((col >= 0) && (col < taille) && !colonnePleine(col)) {
        int ligne = taille-1;
        while (!valide && ligne >=0) {
            if (tab[col][ligne] == 0) {
                tab[col][ligne] = joueur;
                valide = true;
            }
            ligne--;
        }
    }
    return valide;
}
```

ou autre solution

```java
public boolean joueCoup ( int col , int joueur ) {
    boolean valide = !colonnePleine (col);
    if (valide) {
        int ligne = taille-1;
        while(tab[col][ligne]!=0)
            ligne--;
        tab[col][ligne] = joueur;
    }
    return valide;
}
```

Test de la valeur de col entre 0 et taille (facultatif, le test peut être fait dans la méthode main)
Test de la colonne non pleine
Initialisation de la ligne à la valeur (taille-1)
Attribution de la valeur de la case visée avec le numéro du joueur
Renvoie d'un booléen correspondant à l'énoncé

(Q2.2) Les joueurs

Considérons qu'un des deux joueurs soit un humain et que l'autre soit géré par le programme. Ces 2 types de joueurs sont définis à partir d'une classe ancêtre commune : la classe `Joueur` dont le code est fourni ici :

```java
/**
 * La classe représentant un joueur (humain ou ordinateur)
 * Un joueur a un nom et possède une couleur représentée par un entier
 */

public class Joueur {

    public String nom;
    public int couleur;

    /**
     * Le constructeur
     * @param nom le nom du joueur
     * @param couleur la couleur associée au joueur (1=rouge / 2=jaune)
     */
    public Joueur(String nom, int couleur) {
        this.nom = nom;
        this.couleur = couleur;
    }

    /**
     * Cette méthode joue un coup avec le tableau reçu en paramètre.
     * @param jeuEnCours la grille avec laquelle jouer.
     * @param col le numero colonne choisi par le joueur
     */
    public boolean joue(Grille jeuEnCours, int col){
        return (jeuEnCours.joueCoup(col, couleur));
    }

}
```

(Q2.2).1 Le joueur humain

Donner le code de la classe `Humain` qui hérite de la classe `Joueur`.
Voici le code correspondant :

```java
/**
 * La classe Humain qui descend de Joueur
 */

public class Humain extends Joueur {

    /**
     * Le constructeur
     */
    public Humain(String nom,int couleur) {
        super(nom, couleur);
    }

}
```

Signature la classe correcte (nom, présence de `extends Joueur`)
Le constructeur adéquat avec appel à super
Aucun nouveau attribut

(Q2.2).2 L'adversaire virtuel

Concernant l'adversaire virtuel géré par le programme, on implémentera le comportement le plus simple : le programme jouera toujours dans la première colonne non pleine. Cette partie sera donc gérée dans la classe principale lorsque ce sera le tour de l'adversaire virtuel. Implémenter la classe `Ordi` qui hérite de la classe `Joueur`. On imposera comme nom à ce jouer virtuel le nom `CPU`.
Voici le code correspondant :

```java
/**
 * La classe Ordi qui descend de Joueur
 */

public class Ordi extends Joueur {

    public Ordi(int couleur) {
        super("CPU", couleur);
    }

}
```

Signature de la classe correcte et appel à super
Le nom du joueur doit être CPU (cf. énoncé)

3 Interface Homme Machine

(Q3.1) Compréhension de code

La figure 5 représente votre écran de résolution 600x600. L'espace entre deux points consécutifs représente 10 pixels. Le point en haut à gauche est donc placé aux coordonnées [10,10]. Le point en bas à droite est donc placé aux coordonnées [590,590]. Représentez sur cette figure (**à rendre avec votre copie**) l'affichage que donnera l'exécution du code suivant (on dessinera soigneusement les bordures des composants utilisés dans le code) :

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

public class IHMDS extends JFrame{

    public static void main(String[] args){
        IHMDS monIHMDS = new IHMDS();
    }

    public IHMDS(){
        super("Ma jolie interface");
        setSize(300,400);
        setLocation(100,50);
        setLayout(null);
        JPanel monPanel1 = new JPanel();
        monPanel1.setBounds(20,20,150,200);
        monPanel1.setLayout(null);
        JButton monBouton1 = new JButton();
        monBouton1.setText("Y");
        monBouton1.setBounds(0,0,50,100);
        JButton monBouton2 = new JButton("GO");
        monBouton2.setBounds(50,100,100,100);
        monPanel1.add(monBouton1);
        monPanel1.add(monBouton2);
        JPanel monPanel2 = new JPanel();
        monPanel2.setBounds(0,0,getWidth(),getHeight());
        monPanel2.setLayout(null);
        monPanel2.add(monPanel1);
        JPanel monPanel3 = new JPanel();
        monPanel3.setBounds(180,10,100,100);
        monPanel3.setLayout(null);
        JButton monBouton3 = new JButton("4");
        monBouton3.setBounds(0,0,50,50);
        JLabel mon4 = new JLabel();
        mon4.setText("123456789");
        mon4.setBounds(0,50,100,50);
        monPanel3.add(monBouton3);
        monPanel3.add(mon4);
        monPanel2.add(monPanel3);
        JPanel monPan4 = new JPanel();
        monPan4.setLayout(null);
        monPan4.setBounds(10,250,200,100);
        JTextField monText = new JTextField("Votre nom");
        monText.setBounds(10,10,180,80);
        monPan4.add(monText);
        monPanel2.add(monPan4);
        setContentPane(monPanel2);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
        }
}
```

La correction est sur la dernière page du sujet

(Q3.2) Implémentation

Suite à l'appui sur le bouton sur lequel le texte `GO` est affiché, le texte `123456789` devra être remplacé par `FIN !`. Proposez un code répondant à ce cahier des charges. Si nécessaire, précisez les lignes à modifier et celles qui sont à ajouter/supprimer.

Voici les modifs attendus :

- *ajout de* `import java.awt.event.*;`
- *ajout de* `implements ActionListener`
- *transfert des déclarations du ou des widgets concernés en dehors du constructeur*
- *les déclarations du ou des wifgets concernés ont été retirée du constructeur*
- *ajout d'un écouteur sur le bouton concerné*
- *Implémentation de la méthode* `actionPerformed`*. Le test sur la source de l'événement est facultatif car l'événement est unique.*

Voici le code complet :

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

public class IHM_DS_event extends JFrame implements ActionListener{

    public JButton monBouton2;
    public JLabel mon4;

    public static void main(String[] args){
        new IHM_DS_event();
    }

    public IHM_DS_event(){
        super("Ma jolie interface");
        setSize(300,400);
        setLocation(100,50);
        setLayout(null);
        JPanel monPanel1 = new JPanel();
        monPanel1.setBounds(20,20,150,200);
        monPanel1.setLayout(null);
        JButton monBouton1 = new JButton();
        monBouton1.setText("Y");
        monBouton1.setBounds(0,0,50,100);
        monBouton2 = new JButton("GO");
        monBouton2.setBounds(50,100,100,100);
        monPanel1.add(monBouton1);
        monPanel1.add(monBouton2);
        JPanel monPanel2 = new JPanel();
        monPanel2.setBounds(0,0,getWidth(),getHeight());
        monPanel2.setLayout(null);
        monPanel2.add(monPanel1);
        JPanel monPanel3 = new JPanel();
        monPanel3.setBounds(180,10,100,100);
        monPanel3.setLayout(null);
        JButton monBouton3 = new JButton("4");
        monBouton3.setBounds(0,0,50,50);
        mon4 = new JLabel();
        mon4.setText("123456789");
        mon4.setBounds(0,50,100,50);
        monPanel3.add(monBouton3);
        monPanel3.add(mon4);
        monPanel2.add(monPanel3);
        JPanel monPan4 = new JPanel();
        monPan4.setLayout(null);
        monPan4.setBounds(10,250,200,100);
        JTextField monText = new JTextField("Votre nom");
        monText.setBounds(10,10,180,80);
        monPan4.add(monText);
        monPanel2.add(monPan4);
        monBouton2.addActionListener(this);
        setContentPane(monPanel2);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e){
        if (e.getSource()== monBouton2) {
            mon4.setText("FIN !");
            System.out.println("coucou");
        }
    }
}
```

Nom :
Prénom :
Groupe :

A RENDRE AVEC VOTRE COPIE

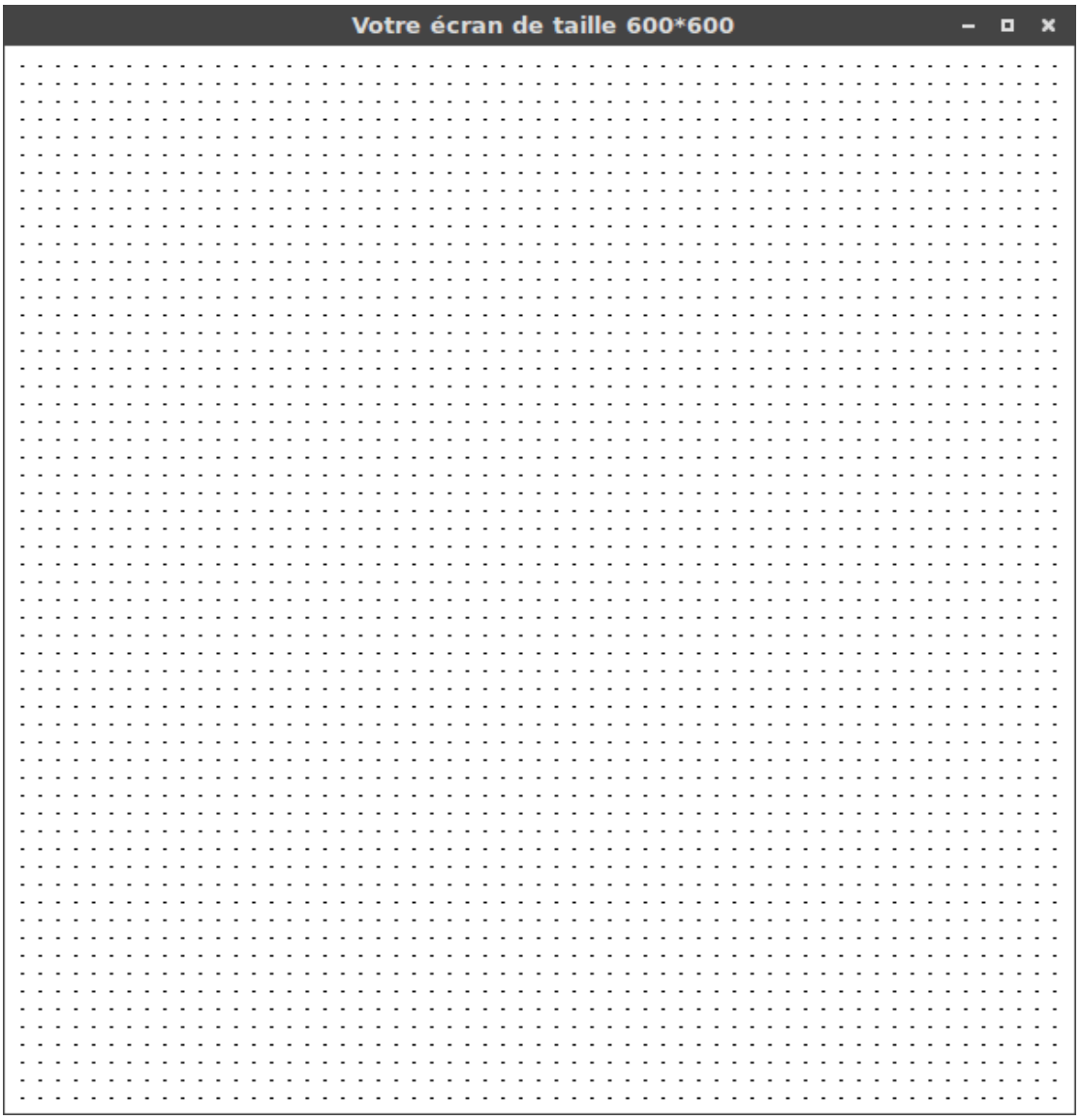


FIGURE 5 – Représentation de l'IHM codée en (Q3.1)

Voici le dessin attendu :

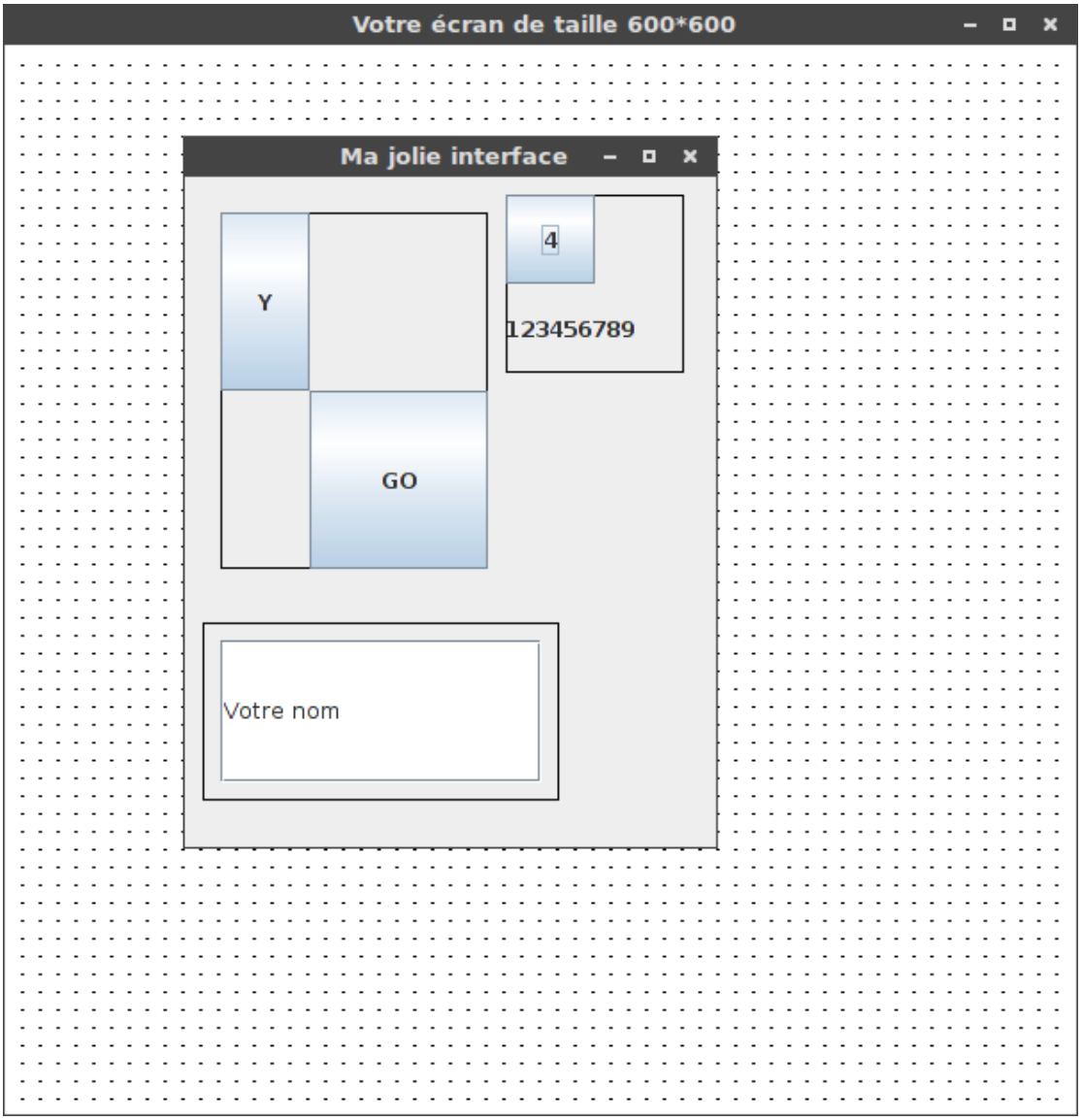


FIGURE 6 – Solution de l'IHM codée en (Q3.1)

Bonne position de l'interface
Bonne position des JPanel et des widgets
Titre de la fenêtre
Texte correct dans chaque widget
Une erreur de ± 10 pixels est autorisée