Durée : 1h30 **AUCUN DOCUMENT - AUCUNE CALCULATRICE**

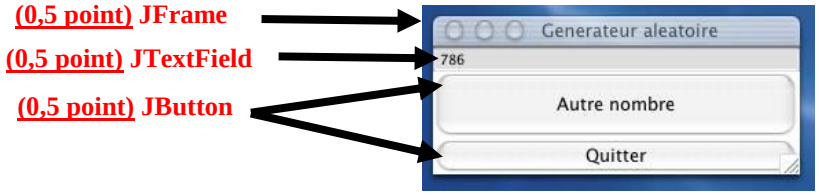
Cette première page à compléter est à joindre à votre copie en indiquant votre nom, prénom et groupe

NOM : ---------- PRENOM : ---------- GROUPE : ----------

Des extraits de la documentation Javadoc sont fournis en ANNEXE C.

Partie 1 : Interface graphique et listeners ≈ 20 minutes *(7 POINTS)*

La maquette suivante définit une fenêtre qui propose une interface graphique permettant de tirer des nombres au hasard, dans l'intervalle [500...800].



Layout utilisé :
(0,5 point) **BorderLayout**

1.1/ (2 points) A l'aide du code ci-dessous, complétez dans la maquette graphique ci-dessus les 3 classes Swing des éléments graphiques fléchés ainsi que le layout **en pointillés**.

1.2/ (2,5 points) Complétez dans le code ci-dessous les **5 lignes soulignées en gras** en vous aidant des **commentaires en gras**.

1.3/ (2,5 points) Complétez le code ci-dessous de sorte à ce que la gestion de l'interaction de l'utilisateur soit prise en compte :

- Un "clic" sur le bouton "Autre nombre" lance un nouveau tirage aléatoire et l'affiche dans le champ texte du nombre obtenu.
- Un "clic" sur le bouton "Quitter" termine l'exécution du programme et ferme la fenêtre.

Si nécessaire, vous préciserez exactement dans quel(s) constructeur(s), méthode(s) et classe(s) votre code sera complété.

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

// La classe Generateur hérite de de la classe JFrame et implémente l'interface ActionListener (0,5 point)
public class Generateur extends JFrame implements ActionListener {

  JTextField nombreTire = new JTextField(); // champ de saisie associé au nombre tiré aléatoirement
  JButton boutonAutre = new JButton("Autre nombre"); // bouton associé à "Autre nombre"
  JButton boutonQuitter = new JButton("Quitter"); // bouton associés à "Quitter"

   public int tirerNombreHasard() {
     // Retourne un nombre aléatoire entre 500 et 800 (0,5 point)
     Random r = new Random(); return r.nextInt(301) + 500;
  }

  public Generateur (){ // constructeur
// Appelle le constructeur hérité de JFrame qui crée une fenêtre avec le titre en paramètre (0,5 point)
     super("Generateur aleatoire");
 // Appelle la méthode tirerNombreHasard() pour modifier le texte dans le champ de saisie nombreTire (0,5 point)
     nombreTire.setText((new Integer (tirerNombreHasard())).toString());
// Positionne le champ de saisie nombreTire en haut de la fenêtre (0,5 point)
     add(nombreTire,"North");

// Suite page suivante pour la question 2.3
```

```
// Ajouter des listeners aux 2 boutons
      boutonAutre.addActionListener(this) ;
      boutonQuitter.addActionListener(this) ;
}


  public void actionPerformed (ActionEvent e) { (2,5 points)
     Object source = e.getSource();

     if (source == boutonAutre)
               nombreTire.setText(tirerNombreHasard() + "");
               // 2ème possibilité : nombreTire.setText(tirerNombreHasard().toString()) ;

     if (source == boutonQuitter) {
               setDefaultCloseOperation (JFrame. EXIT_ON_CLOSE);
               System.exit(0);
      }
  }
}
```

Partie 2 : Diffusion d'une PlayList de vidéos ≈ 70 minutes *(13 POINTS)*

On souhaite mettre en place un logiciel de diffusion d'une PlayList de vidéos. On trouvera deux sortes de vidéos : d'une part des bandes annonces de films, d'autre part des publicités.

Le diagramme de classes proposé en ANNEXE A suit les règles ci-dessous :

- Chaque vidéo a comme propriété commune le titre et la durée;
- Il y a deux sortes de vidéos : les bandes annonces et les publicités ;
- Chaque bande annonce a une note privée qui peut être modifiée ;
- Chaque publicité a une marque privée qui peut être consultée ;
- Les vidéos de la PlayList sont regroupés dans une collection de type **ArrayList** (voir ANNEXE C) ;
- Dans cette PlayList, on peut effectuer les opérations suivantes :
  - rechercher un titre de vidéo dans l'**ArrayList**,
  - calculer la moyenne des notes non nulles des bandes annonces (pas les publicités) de cette **ArrayList**,
  - trier par ordre décroissant des notes des bandes annonces (pas les publicités) de cette **ArrayList**

Des extraits utiles du code de ces classes sont définies est en ANNEXE B. Il est inutile de recopier ces extraits et de connaître les parties du code cachées pour faire les exercices demandés.

Des extraits de la documentation Javadoc sont fournis en ANNEXE C.

Le code à compléter est encadré en gris : lisez attentivement les commentaires pour vous aider. Votre code devra être commenté pour plus de clarté. Lisez attentivement toute cette partie avant de commencer. Vous n'êtes pas obligés de faire les exercices dans l'ordre. Faites au mieux ☺

1) Pour la classe **Vidéo** modélisée dans le diagramme de l'ANNEXE A et définie en ANNEXE B.1 :

a) Quel est l'intérêt qu'elle soit abstraite ? ***(0,5 POINT)***

Cette classe non instanciable permet de factoriser les propriétés communes de ses classes filles BandeAnnonce et Publicité

b) Quel est l'intérêt de mettre ses attributs en visibilité « **protected** » ? ***(0,5 POINT)***

Pour les rendre visibles à ses classes filles uniquement

2) Dans la classe **Vidéo** définie en ANNEXE B.1, complétez le code de son constructeur qui initialise tous ses attributs avec ses paramètres. ***(1 POINT)***

```
public Vidéo (String titre, int durée) {this.titre=titre ; this.durée=durée ;}
```

3) Dans la classe **PlayList** définie en ANNEXE B.2 :

a) Complétez la méthode **rechercher** en vous servant des commentaires. ***(2 POINTS)***

```
public static Video rechercher (String titre) throws VideoInexistante {
    Vidéo v = null ; // vidéo à rechercher
    boolean trouve = false ; // booléen indiquant si la vidéo existe
      int i = 0 ; // indice de la liste

    // boucle de recherche dans la liste
    while (i< liste.size() && !trouve) {
            v = liste.get(i);
            if (v.getTitre() == titre) // pas v.titre car titre est protected dans la classe Vidéo
                    trouve = true;
            i++;
    }

    if (!trouve)
            throw new VideoInexistante() ; // vidéo non trouvée : exception créée
    else
            return v ; // retourner la vidéo trouvée
}
```

b) Complétez sa méthode **moyenne** en vous servant des commentaires. ***(2 POINTS)***

```
public float moyenne () throws NotesNulles {
    int j = 0 ; // nombre de notes non nulles
    float note, somme = 0 ; // somme des notes

    // calcul de la somme des notes non vides
    for (int i=0; i< liste.size() ; i++) {
            Video v = liste.get(i); note = 0;
            if (v instanceof BandeAnnonce)
                    note = ((BandeAnnonce)v).getNote() ; // pas liste.get(i).note  car note est private
            if (note != 0) { // si la note n'est pas nulle, l'ajouter à la somme et incrémenter nombre notes
                    somme += note ; j++ ;
            }
    }

    if (j==0)
            throw new NotesNulles() ; // toutes les notes sont nulles : exception créée
    else
            return somme/j ; // retourner la moyenne des notes non vides
}
```

c) Codez la méthode qui trie les bandes annonces (pas les publicités) de la playlist par ordre décroissant de notes. Puis elle affiche le ~~nom~~ titre, ~~le prénom~~ la durée, la note de chaque bande annonce de la palylist triée, ainsi que la moyenne (voir question précédente) des notes non nulles de cette playlist. ***(3 POINTS)***

```
public void trier() {
   ArrayList<BandeAnnonce> listetriee = new ArrayList<BandeAnnonce>(); // liste à trier
   Vidéo v1, v2 ;
   float note1, note2 ;

   // Ajouter des bandes annonces dans la liste à trier
   for (int i=0 ; i < liste.size() ; i++) {
     Vidéo v = liste.get(i) ;
     if (v instanceof BandeAnnonce)
            listetriee.add((BandeAnnonce) v) ; // ajouter la bande annonce dans la liste à trier
   }

   // Trier la liste à trier par note décroissante
   for (int i=0 ; i < listetriee.size()-1 ; i++)
     for (int j=i+1 ; j < listetriee.size() ; j++) {
            // Récupérer chaque paire de vidéos pour comparer leur note
            v1 = listetriee.get(i) ; v2 = listetriee.get(j);
            note1 = v1.getNote(); note2 = v2. getNote();
            if (note2 > note1) {
                    // Permuter les 2 vidéos dans la liste à trier
                    listetriee.set(i, v2) ;
                    listetriee.set(j, v1) ;
            }
     }

   // Afficher le titre, la durée, la note de chaque bande annonce de la liste triée
   for (int i=0 ; i < listetriee.size() ; i++) {
     Vidéo v = listetriee.get(i);
     System.out.println(v.getTitre()+" ; "+ v.getDurée()+" ; "+ v.getNote());
   }

   // Afficher la moyenne des notes non nulles de la liste triée sauf toutes les notes sont nulles
   try { liste= listetriee ; System.out.println("Moyenne des notes non nulles="+ moyenne()) } ;
   catch (NotesNulles n) { System.out.println("Toutes les notes sont nulles") };
}
```

4) Dans la classe **BandeAnnonce** définie en ANNEXE B.3 :

a) Constructeur à compléter qui initialise par héritage les attributs de l'objet référencé avec ses paramètres (voir ANNEXE B.1). ***(1 POINT)***

```
public BandeAnnonce (String titre, int durée, float note) {
      super(titre, durée) ; this.note = note ;
}
```

b) Complétez sa méthode **setNote** en vous servant des commentaires. ***(2 POINTS)***

```
public void setNote (float note, String titre, PlayList liste) {
      this.note = note ;

      try { Vidéo v = liste.liste.rechercher(titre) ; }

      catch(VideoInexistante vi) {
            liste.liste.add(new BandeAnnonce(titre, 0, note)) ;
            // 2ème possibilité : liste.liste.add(this) ;

      }
}
```

5) Dans la classe **Publicité** définie en ANNEXE B.4 :

a) Codez l'accesseur en lecture **getMarque** permettant de consulter l'attribut privé **marque**. ***(0,5 POINT)***

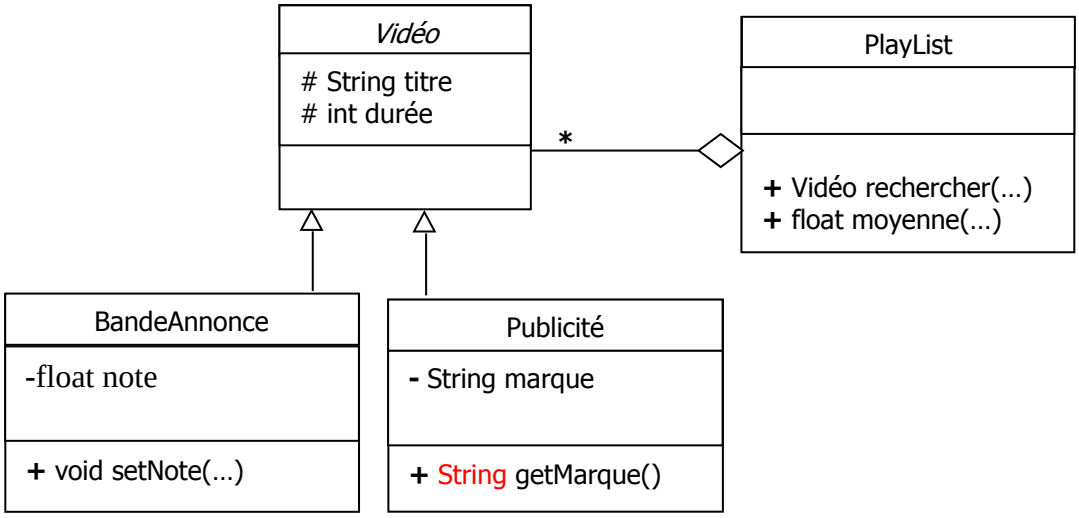
```
public String getMarque () {
      return this.marque ;
}
```

b) Pourquoi ne faut-il pas mettre l'accesseur en écriture pour l'attribut privé **marque** ? ***(0,5 POINT)***

Pour ne pas pouvoir modifier la marque de la publicité

ANNEXE A : DIAGRAMME DE CLASSES

Les classes en italique représentent des classes abstraites.
Les niveaux de visibilité sont : # protected, - private ou + public



ANNEXE B : EXTRAITS DU CODE

ANNEXE B.1
La classe **Vidéo** est une classe abstraite définie comme suit :

```
abstract class Vidéo {
// Attributs de niveau de visibilité « protected »
  protected String titre;
  protected int durée;

// Constructeur qui initialise tous les attributs de la vidéo référencée avec les paramètres
  public Vidéo (String titre, int durée) {…}
}
```

ANNEXE B.2
Un extrait de la classe **PlayList** est défini ci-dessous :

```
import java.util.ArrayList ;

public class PlayList {
// Attributs publics
public static ArrayList<Vidéo> liste ; //  liste publique statique des vidéos

/**
Méthode statique à compléter qui recherche une vidéo dans l'ArrayList ci-dessus à partir de son titre en paramètre. Si la vidéo recherchée existe, cette méthode retourne la vidéo trouvée.  Si cette vidéo n'existe pas, l'exception VideoInexistante est créée.
*/
  public static Video rechercher (String titre) throws VideoInexistante { …  }


/**
Méthode à compléter qui calcule et retourne la moyenne des notes non nulles des vidéos de la playlist référencée. Si toutes les notes sont nulles, l'exception NotesNulles est créée
*/
  public float moyenne () throws NotesNulles { … }

}
```

ANNEXE B.3
Un extrait de la classe **BandeAnnonce** est défini ci-dessous :

```
public class BandeAnnonce extends Vidéo {
// attributs privés
   private float note ; // note donnée à la bande annonce

// Constructeur à compléter qui initialise par héritage les attributs de l'objet référencé avec ses paramètres (voir ANNEXE B.1)
  public BandeAnnonce (String titre, int durée, float note) {  ... }

/**
 Cette méthode modifie d'abord la note de la bande annoncé référencée avec celle en paramètre. Puis elle recherche le titre en paramètre dans la liste en paramètre en appelant la méthode rechercher(...) de l'ANNEXE B.2 . Si cette bande annonce n'existe pas, l'exception VideoInexistante déclenchée ajoute cette bande annonce dans la liste
*/
  public void setNote (float note, String titre, PlayList liste) {...}

}
```

ANNEXE B.4
Un extrait de la classe **Publicité** est défini ci-dessous :

```
public class Publicité extends Vidéo {
// attributs privés
   private String marque ; // marque vendue par la publicité

}
```

ANNEXE C : extraits de la documentation Javadoc

javax.swing
Class JButton

Method Summary	
String	**getText**() Returns the button's text.
void	**setText**(String text) Sets the button's text
void	**addActionListener**(ActionListener l) Adds an ActionListener to the button.

Class JFrame

Constructor Summary
JFrame(String title) Creates a new, initially invisible Frame with the specified title.

Field Summary	
static int	**EXIT_ON_CLOSE** The exit application default window close operation.

Method Summary	
Container	**getContentPane**() Returns the contentPane object for this frame.
void	**setContentPane**(Container contentPane) Sets the contentPane property.
void	**setDefaultCloseOperation**(int operation) Sets the operation that will happen by default when the user initiates a "close" on this frame.
void	**setSize**(int width, int height) Resizes this component so that it has width width and height height.
void	**setVisible**(boolean b) Shows or hides this Window depending on the value of parameter b.

java.awt
Class Container

Method Summary	
Component	**add**(Component comp) Appends the specified component to the end of this container.

java.awt.event
Interface ActionListener

Method Summary	
void	**actionPerformed**(ActionEvent e) Invoked when an action occurs.

java.util

Class ArrayList

Constructor Summary
ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(Collection<? extends E> c) Constructs a list containing the elements (objects) of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
boolean	**add**(E e) Appends the specified element (object) to the end of this list.
void	**add**(int index, E element) Inserts the specified element (object) at the specified position in this list.
boolean	**contains**(Object o) Returns true if this list contains the specified element (object)
E	**get**(int index) Returns the element (object) at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element (object) in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements (objects).
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element (object).
int	**size**() Returns the number of elements (objects) in this list.

Class EventObject

Field Summary	
protected Object	**source** The object on which the Event initially occurred.

Method Summary	
Object	**getSource**() The object on which the Event initially occurred.

Class Random

Constructor Summary
Random() Creates a new random number generator.

Method Summary	
int	**nextInt**(int n) Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.