Nom Prénom :

Consignes relatives au déroulement de l'épreuve

Date :	16 juin 2017
Contrôle de :	**3IRC - Module « Programmation Orienté Objet » -**
Durée :	2h
Professeur responsable :	Françoise Perrin
Documents :	Supports de cours et TP autorisés Livres, calculatrice, téléphone, ordinateur et autres appareils de stockage de données numériques interdits.
Divers :	Les oreilles des candidats doivent être dégagées.

Rappels importants sur la discipline lors des examens

La présence à tous les examens est strictement obligatoire ; tout élève présent à une épreuve doit rendre une copie, même blanche, portant son nom, son prénom et la nature de l'épreuve.
Une absence non justifiée peut entrainer l'invalidation du module
Toute suspicion sur la régularité et le caractère équitable d'une épreuve est signalée à la direction des études qui pourra décider l'annulation de l'épreuve ; tous les élèves concernés par l'épreuve sont alors convoqués à une épreuve de remplacement à une date fixée par le responsable d'année.
Toute fraude ou tentative de fraude est portée à la connaissance de la direction des études qui pourra réunir le Conseil de Discipline. Les sanctions prises peuvent aller jusqu'à l'exclusion définitive du (des) élève(s) mis en cause.

Recommandations

L'objectif de cette épreuve est de vérifier votre capacité à comprendre un programme existant et à le maintenir, et en particulier, vérifier que vous êtes capable :

De justifier ou critiquer certains choix de conception objet.
De justifier ou critiquer le choix de certaines structures de données.
De dire ce que font certaines instructions en expliquant leur intérêt pour le programme.
D'écrire des portions de code inexistantes.

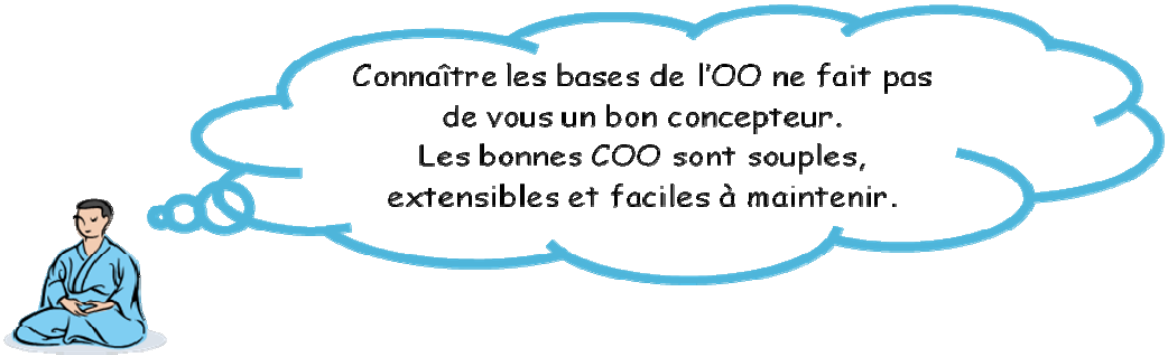
Lisez bien tout l'énoncé - y compris les annexes et trucs et astuces -

Les questions se rapportent au projet de jeu d'échec réalisé en séance

Les questions sont écrites en gras.

Pour les instructions Java, la syntaxe n'a pas d'importance pourvu qu'elle soit compréhensible.

Pour les questions qui attendent des réponses en français, ne récitez (ne recopiez ☹) pas le cours de manière théorique mais soyez **très près** de l'exemple. Soyez **synthétique** mais **précis**.



Rappels de conception du projet

La conception du projet a permis d'identifier un certain nombre de classes « métier » et en particulier :

- Des pièces : roi, reine, fous, pions, cavaliers, tours. Il existe 16 pièces blanches et 16 pièces noires.
- Des jeux : ensemble des pièces d'un joueur. Il existe 1 jeu blanc et 1 jeu noir.
- 1 échiquier qui contient les 2 jeux.

Chaque classe a ses propres responsabilités. Ainsi la classe `Jeu` est responsable de créer ses `Pieces` et de les manipuler. `L'Echiquier` quant à lui crée les 2 jeux mais ne peut pas manipuler directement les pièces. Pour autant, c'est lui qui est capable de dire si un déplacement est légal, d'ordonner ce déplacement, de gérer l'alternance des joueurs, de savoir si le roi est en échec et mat, etc. Pour ce faire, il passe donc par les objets `Jeu` pour communiquer avec les `Pieces`.

Les classes sont donc parfaitement bien encapsulées et les seules interactions possibles avec une IHM se font à travers `L'Echiquier` et en aucun cas une IHM ne pourra directement déplacer une `Pieces` sans passer par les méthodes de `L'Echiquier` (en fait à travers une classe `ChessGame`). L'architecture de l'application est donnée en annexe.

1. Questions de programmation sur le projet (5 points)

a) Soit la classe Coord suivante :

```
public class Coord {

   public int x, y;

   public Coord(int x, int y) {
         this.x = x;
         this.y = y;
   }
   public String toString() {
         return "[x=" + x + ", y=" + y + "]";
   }
   public static boolean coordonnees_valides(int a, int b){
         return ( (a<=7) && (a>=0) && (b<=7) && (b>=0) );
   }
   public int hashCode() {
         // …
   }
   public boolean equals(Object obj) {
         // …
   }
}
```

Pourquoi, alors qu'on nous a répété qu'il fallait protéger les attributs d'une classe (avec le qualificateur « private ») les attributs x et y sont-ils publics (Ne répondez pas que c'est une erreur ☺) **?**

b) Soit les interface et classes `Pieces`, `AbstractPiece` et toutes les déclinaisons de `Pieces` (illustrées par la classe `Tour`) suivantes :

```
public interface Pieces {
   public int getX();
   public int getY();
   public Couleur getCouleur();
   public String getName();
   public boolean isMoveOk(int xFinal, int yFinal) ;
   public boolean move(int xFinal, int yFinal);
   public boolean capture();
};
public abstract class AbstractPiece implements Pieces {
   private int x, y;
   private Couleur couleur;
   public AbstractPiece(Couleur couleur, Coord coord){
          this.x = coord.x;
          this.y = coord.y;
          this.couleur=couleur;
   }
   public int getX(){
          return this.x;
   }
   public int getY(){
          return this.y;
   }
   public Couleur getCouleur(){
          return this.couleur;
   }
   public String getName(){
          return this.getClass().getSimpleName();
   }
   public boolean move(int x, int y){
          boolean ret = false;
          if(Coord.coordonnees_valides(x,y)){
                this.x=x; this.y=y;
                ret = true;
          }
          return ret;
   }
   public boolean capture(){
          this.x=-1; this.y=-1;
          return true;
   }
   public String toString(){
          String S = (this.getClass().getSimpleName()).substring(0, 2)
                       + " " + this.x + " " + this.y;
          return S;
   }
   public abstract boolean isMoveOk(int xFinal, int yFinal) ;
}
public class Tour extends AbstractPiece {
   public Tour(Couleur couleur_de_piece, Coord coord) {
          super(couleur_de_piece, coord);
   }
   public boolean isMoveOk(int xFinal, int yFinal) {
          boolean ret = false;
          if ((yFinal == this.getY()) || (xFinal == this.getX())) {
                ret = true;
          }
          return ret;
   }
}
```

1. **Pourquoi dans la classe `AbstractPiece` les attributs x et y sont-ils déclarés avec le qualificateur « private » ?**

2. **Sachant que la méthode `isMoveOk()` dans les classes dérivées s'appuie sur les positions initiales de la pièce, auraient-on pu / dû déclarer x et y avec le qualificateur « protected ». Justifiez.**

3. **Pour indiquer qu'une pièce a été capturée, on a choisi de mettre ses coordonnées à -1 dans la méthode `capture()`. Proposez une autre manière de faire** (en français ou en Java).

4. **Dans le constructeur de la classe `Tour` à quoi sert l'instruction**
   `super(couleur_de_piece, coord);` **?**

2. Maintenance évolutive du projet (7 points)

a) Dans la classe `Echiquier` (description en annexe), **écrire une méthode qui retourne la liste des coordonnées des cases sur laquelle une pièce aux coordonnées x, y peut être déplacée :**

```
public List<Coord> getCoordPieceMoveOk(int x, int y) ;
```

b) Est-ce qu'on aurait pu stocker ces coordonnées dans un Set plutôt que dans une List ? Justifiez. Si oui, est ce qu'il aurait fallu modifier la classe Coord ? Justifiez.

c) Dans la classe Jeu écrire une méthode privée de promotion d'un pion en une pièce d'un type passé en paramètre : `private void pawnPromotion(int x, int y, String type) ;`

Lorsqu'un pion atteint la dernière rangée, il doit être changé (promu) en une pièce de sa couleur qui peut être au choix une dame, une tour, un fou ou un cavalier (info contenue dans le type passé en paramètre). Attention, il ne relève pas de la responsabilité de cette méthode de savoir si la promotion est à effectuer, ni si la pièce aux coordonnées x,y est bien un pion, mais seulement d'effectuer cette promotion.
Pour mémoire :

- La classe `Jeu` a comme attribut une liste de `Pieces`.
- Il existe une fonction `newPiece` qui permet de fabriquer une pièce d'un type donné dans la classe ChessSinglePieceFactory (paramètres : couleur, type, x, y)

3. Réflexions sur la conception du projet (8 points)

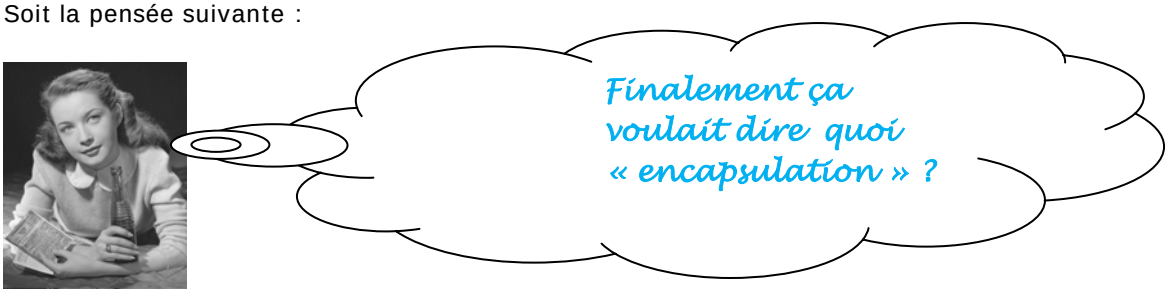
A la lumière de la conception du projet (rappel diagramme UML en annexe),

a) A quoi sert l'interface `Pieces` (votre réponse doit permettre de répondre à la question plus générale « à quoi sert une interface ») **?**

b) A quoi sert la classe `AbstractPieces` (votre réponse doit permettre de répondre à la question plus générale « à quoi sert une classe abstraite ») ?

c) L'affichage d'une couleur ou d'un cadre ou autre sur les cases de destination possibles lorsque l'utilisateur sélectionne une pièce (un JLabel en fait) relève de la responsabilité de la méthode `mousePressed()` de la GUI ; vous étiez tous d'accord sur ce point. En revanche, pour la recherche de la liste des cases (au sens coordonnées x, y cette fois) certains ont confié cette responsabilité à la vue, d'autres au contrôleur, d'autres au modèle (classe `Echiquier`, avec propagation des appels depuis la vue en passant par le contrôleur et la classe `Chessgame`). **Expliquez pourquoi cette dernière option est celle qui devait être mise en œuvre et pourquoi.**

d) Soit la pensée suivante :



Expliquez en quoi la conception du projet répond à la question (illustre le concept d'encapsulation)**.**

Annexe Interface List<E>

Method Summary

boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).
boolean	**remove**(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
	...

Annexe Interface Set<E>

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns `true` if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Annexe Class HashSet<E>

Constructor Summary

`HashSet()`
Constructs a new, empty set; the backing `HashMap` instance has default initial capacity (16) and load factor (0.75).

`HashSet(Collection<? extends E> c)`
Constructs a new set containing the elements in the specified collection.

…

Method Summary

…

Annexe Class TreeSet<E>

Constructor Summary

`TreeSet()`
Constructs a new, empty tree set, sorted according to the natural ordering of its elements.

`TreeSet(Collection<? extends E> c)`
Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.

`TreeSet(Comparator<? super E> comparator)`
Constructs a new, empty tree set, sorted according to the specified comparator.

`TreeSet(SortedSet<E> s)`
Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

…

Method Summary

…

Annexe Class Echiquier

java.lang.Object

model.Echiquier

All Implemented Interfaces:

BoardGames

---

public class **Echiquier**
extends java.lang.Object
implements BoardGames

Author:

francoise.perrin

Constructor Summary

Echiquier()

Method Summary

Couleur	**getColorCurrentPlayer**()
java.lang.String	**getMessage**()
Couleur	**getPieceColor**(int x, int y)
java.util.List<PieceIHMs>	**getPiecesIHM**()
boolean	**isEnd**()
boolean	**isMoveOk**(int xInit, int yInit, int xFinal, int yFinal) Permet de vérifier si une piéce peut être déplacée.
static void	**main**(java.lang.String[] args)
boolean	**move**(int xInit, int yInit, int xFinal, int yFinal) Permet de déplacer une piéce connaissant ses coordonnées initiales vers ses coordonnées finales.
void	**switchJoueur**() Permet de changer le joueur courant.
java.lang.String	**toString**()

Annexe : architecture globale de l'application