Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Carthage

Ecole Nationale d'Ingénieurs de Carthage

ENICARTHAGE

المدرسة الوطنية للمهندسين بقرطاج

Ecole Nationale d'Ingénieurs de Carthage

وزارة التعليم العالي و البحث العلمي

جامعة قرطاج

المدرسة الوطنية للمهندسين بقرطاج

A.U. : 2020 /2021

Section :**2Année Ingénieur**

Module : **Programmation Java** DS : 26 - 11 – **2020**

Durée : 1 heure **Documents : Non autorisés** **Nombre de pages : 06**

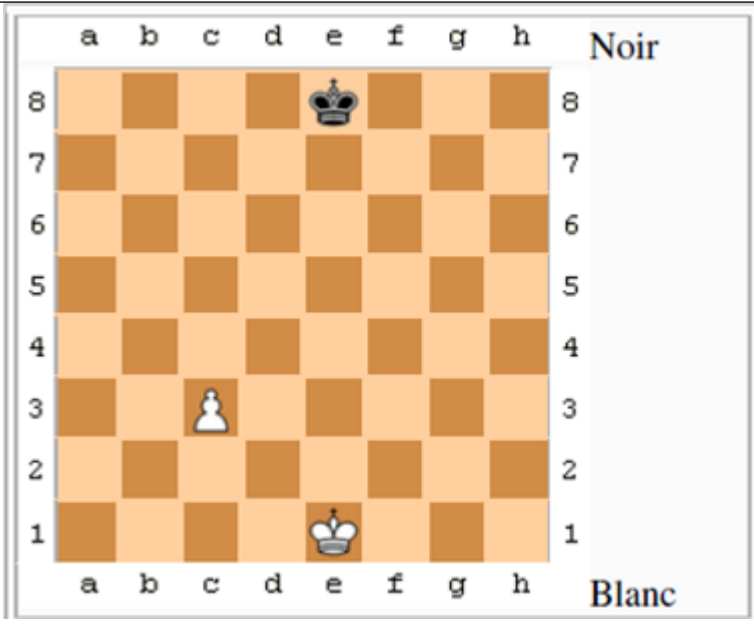
Il sera tenu compte de la clarté, de la présentation des réponses.

Nom …….. Prénom …… Groupe……CIN…….

Problème:

L'objectif de ce problème est d'implémenter en Java les classes, interfaces et manipulations de base d'un jeu des échecs. On s'intéressera qu'aux déplacements des rois. Ce jeu se joue à deux joueurs (un avec des pièces noires et l'autre avec des pièces blanches) sur un plateau carré de 8 cases sur 8, appelé échiquier.

Les colonnes de l'échiquier portent les lettres de **a** à **h**, et les lignes les chiffres de **1** à **8**. La case **a1** est par convention la case la plus à gauche de la première ligne vu du côté du joueur Blanc (en bas).



Exemple : Roi noir en e8, Roi blanc en en e1, pion blanc en c3

Compléter le code Java en suivant les commentaires ainsi que les directives suivantes :

1) Une case dans un échiquier est définie par ses coordonnées (colonne, ligne) toutes les deux de type **int** variant entre 0 et 7. Par exemple la case a1 est aux coordonnées (0,0). Elle est munie d'une couleur, qu'on représentera par le booléen isBlanc, vrai si la case est blanche. Chaque Case peut être vide ou contenir au plus une Pièce.
2) On définit les pièces de jeu par la classe abstraite implémentée par la classe abstraite définie au début du code.
3) Le Roi est une pièce qui peut se déplacer dans n'importe quelle direction (y compris en diagonale) d'une seule case. Il peut manger une pièce adverse en se plaçant sur la case où elle se trouve. La pièce ainsi mangée est retirée de la partie. Il ne peut pas manger une pièce de sa couleur.
4) Le dessin des pièces revient à afficher leurs symboles sur écran par utilisation « \u » suivi du code hexadécimal (voir annexe1).

CODE JAVA

public abstract class Piece

{ boolean estBlanc;

public Piece(boolean estBlanc) {

this.estBlanc = estBlanc; }

public boolean estBlanc() {

return estBlanc; }

public abstract boolean deplacementValide(Case ori, Case dest);}

/* Implémentation d'une interface fonctionnelle Dessin avec une seule méthode dessiner()/*

.....

.....

.....

/* **Implémentation de la classe Case** avec déclanchement d'une erreur de construction*/

public class Case {

private boolean isBlanc;

private int ligne, colonne ;

private Piece contenu;

public Case (int ligne, int colonne,boolean isBlanc).....

.....

.....

.....

.....

.....

.....

// getters et setters

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

/* méthode d'affichage standard qui ne retourne que les attributs de la position*/

.....

.....

.....

.....

/* **Implémentation de la classe correspondant à l'erreur de construction**. On doit afficher un message d'erreur, ainsi que la position erronée par passage d'objet*/

.....

.....

.....

.....

.....

.....

// Implémentation de la classe Roi

public class Roi .....

.....

/* attribut de la case courante dans laquelle se trouve le roi*/

private Case courante;

public Roi(boolean estBlanc) {

.....

.....

/* Implémentation de la méthode dessiner() définie dans l'interface fonctionnelle de Dessin/*

.....

.....

.....

.....

.....

/* Implémentation d'une méthode de déplacement du roi héritée*/

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Case getCase(){return courante;}

void setCase(Case courante)
{this.courante=courante;}.

.....

.....

/* méthode d'affichage standard qui ne retourne que les attributs de la position*/

.....

.....

.....

.....

/* Implémentation de la classe correspondant à l'erreur de construction. On doit afficher un message d'erreur, ainsi que la position erronée par passage d'objet*/

.....

.....

.....

.....

.....

.....

// Implémentation de la classe principale

public class Principal {

/* Méthode statique d'initialisation d'un échiquier qui permet de le retourner sous forme d'un tableau bidimensionnel*/

.....

Case[][]echiquier = new Case[8][8];

.....

.....

.....

if(i+j%2==0) echiquier [i][j]=.....

.....

.....

.....

.....

.....

/* Méthode qui permet tester l'interface de Dessin par expression Lambda, et ce en dessinant un cavalier noir */

public void Tester(){

.....

.....

.....}

// Méthode main

.....

.....

// Appel d'initialisation de l'échiquier

Case [][] ech = new Case[8][8];

.....

Roi rb = new Roi(true); rb.setCase(ech[4][0]);

Roi rn = new Roi(true); rn.setCase(ech [4][7]);

Scanner sc = new Scanner(System.in);

int i =0; boolean b = true;

int ligne = 0; int colonne = 0;

// Création de deux listes chainées

/* liste1 (resp. liste2) pour enregistrer les cases de déplacement de rb (resp. de rn)*/

.....

.....

/* Lancer un jeu de déplacement juste des rois comme suit :

- ✓ afficher la couleur du joueur,
- ✓ puis donner la main au joueur concerné pour qu'il insère les coordonnés de la position de la case de destination via le clavier
- ✓ En cas d'un mouvement valide on enregistre la case dans la liste adéquate, et on effectue la mise à jour de la case courante du roi
- ✓ Idem pour le deuxième joueur
- ✓ Lorsque les deux joueurs terminent une étape du jeu on affiche un message pour pouvoir quitter ou continuer le jeu ; ceci donne la possibilité de sortir de la boucle du jeu à chaque étape */

while (b) {

if (i%2==0){

System.out.println("joueur blanc");

System.out.println("donner les coordonnées deux entiers entre 0 et 7 comme coordonnées de la case destination");

.....

.....

.....

.....

.....

else { System.out.println("joueur noir");

System.out.println("donner les coordonnées deux entiers entre 0 et 7 comme coordonnées de la case destination");

.....

.....

.....

.....

.....

.....}

System.out.println("pour rester taper true ");

System.out.println("pour sortir taper false ");

.....

.....

// Pour le reste n'utiliser que des streams et la méthode ForEach () pour l'affichage

/* Première manip : en une seule ligne de code, afficher pour chaque case de la liste1 la somme de ses coordonnées*/

.....

/* Deuxième manip : en une seule ligne de code, compléter l'affichage du nombre des cases dans liste1 dont le numéro de ligne est entre 0 et 3. Utiliser mapToInt.

System.out.println(.....

.....) ;

/* Troisième manip : en une seule ligne de code, déterminer l'ensemble des cases dans liste2 dont le numéro de colonne est impair*/

Set<Case> ensemble = new HashSet();

ensemble =.....

..... ;

Compléter sur une autre feuille si l'espace est insuffisant

Annexe 1

Les symboles d'échecs font partie de la table des caractères Unicode :

Nom	**symbole**	**Code hexadécimal Unicode**	**Nom**	**symbole**	**Code hexadécimal Unicode**
Roi Blanc		2654	Roi Noir		265A
Dame blanche		2655	Dame Noire		265B
Tour blanche		2656	Tour Noire		265C
Cavalier blanc		2658	Cavalier noir		265E
Pion blanc		2659	Pion noir		265F

Annexe 2

java.util.stream.Collectors(méthodes statiques)
toMap(Function<?superT,?extendsK>, Function<?superT,?extendsV>)): retourne une sorte de Map<K,V> toSet() : retourne une sorte de Set<T> toList() : retourne une sorte de List<T>
Interfaces fonctionelles + leurs méthodes abstraites
Function <T,U>U apply (T) UnaryOperator <T>T apply (T) Consumer <T> void accept (T) Supplier <T>T get() Predicate <T>boolean test (T) Comparator<T>int compare(T,T)

java.util.Scanner	Object	java.util.StringTokinezer	String
Scanner(System.in); String next() String nextLine() boolean nextBoolean() int nextInt() double nextDouble() boolean hasNext() boolean hasNextLine() boolean hasNextBoolean() boolean hasNextInt() boolean hasNextDouble()	String toString() boolean equals() finalize() clone()	StringTokenizer (String) StringTokenizer (String, String) StringTokenizer (String , String, boolean) String nextToken() boolean hasMoreTokens() int countTokens()	int length() int indexOf(char , int) String substring(int, int) boolean contains(String) boolean equals (String)
Exception	**Integer**		
String getMessage()	int parseInt(String) : static		
	Class	**StringBuffer**	**Math**
	String getName() Class getSuperclass() String toString() newInstance()	int length() int capacity() StringBuffer append(type de base) StringBuffer insert(int, type de base) StringBuffer reverse()	PI: static double sqrt(double): static double random(): static

Interface Collection<T> java.util.Collection	Interface List <T> java.util.List	Interface Map <K,V> Java.util.Map
int size() boolean isEmpty() boolean contains(T) add(T) remove(T) clear() T[] toArray() Iterator<T> iterator() forEach(Consumer<?superT>)) stream()	add(int, T) set(int,T) T get(int) remove(int) indexOf(T) lastIndexOf(T) subList(int,int) ListIterator<T> listIterator()	put(K, V) V get(K) String remove(K) boolean remove (K, V) boolean containsKey(K) boolean containsValue(V) int size() boolean isEmpty() putAll(Map) clear() Set<K> keySet(); Collection<V> values() Set<Entry<K,V>> entrySet() Entry: static
java.util.Iterator	**java.util.ListIterator**	
boolean hasNext() T next() remove()	boolean hasNext() T next() int previousIndex() ; remove() boolean hasPrevious() T previous()	

Stream<T>: java.util.Stream	
Stream<T> of(T…values): static Stream <T> filter(Predicate<?superT>) Stream<R> map(Function<?superT,?extendsR>) IntStream mapToInt(ToIntFunction<?superT>) DoubleStream mapToDouble(ToDoubleFunction<?superT>) LongStream mapToLong(ToLongFunction<?superT>) forEach(Consumer<?superT>) forEachOrdered(Consumer<?superT>)	Optional <T> reduce(BinayOperator<T>) Stream<T> limit(long) long count() max(Comparator<?superT>) min(Comparator<?superT>) Stream<T> sorted(Comparator<?superT>) Object[] toArray() R Collect(Collector<?super T, A, R>)