Nom - Prénom :

## Consignes relatives au déroulement de l'épreuve

| | |
|---|---|
| Date : | 15 juin 2018 |
| Contrôle de : | **3IRC - Module « Programmation Orienté Objet » - DS2 1ère session** |
| Durée : | 2h |
| Professeur responsable : | Françoise Perrin |
| Documents : | Supports de cours et TP autorisés<br>Livres, calculatrice, téléphone, ordinateur et autres appareils de stockage de données numériques interdits. |

## Rappels importants sur la discipline lors des examens

Les oreilles des candidats doivent être dégagées.

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, de 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 trucs et astuces – avant de commencer à répondre, certaines questions pouvant apporter des éléments de réponses aux questions précédentes.**

**Les questions sont écrites en gras et référencent pour la plupart une portion du programme.**

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

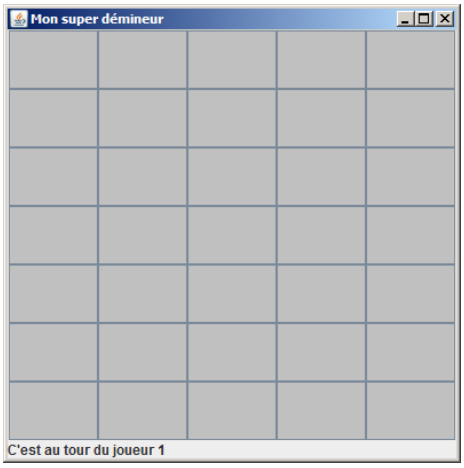
Pour les questions qui attendent des réponses en français, ne récitez (ne recopiez ☹) pas le cours de manière théorique mais soyez **très près** de l'exemple. Soyez **synthétique** mais **précis**.

N'oubliez pas d'écrire votre nom sur le sujet si vous séparez la partie code de la partie questions (page 9) ☺.

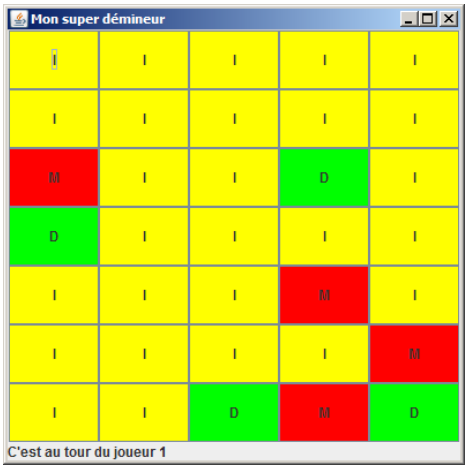
## Présentation du programme

Il s'agit de réaliser un programme multijoueur ressemblant au jeu du démineur. Pour simplifier, seront codés en dur le nombre de joueurs, de lignes et de colonnes. En cas de multijoueur, tous jouent sur le même écran (1 seul programme s'exécute).

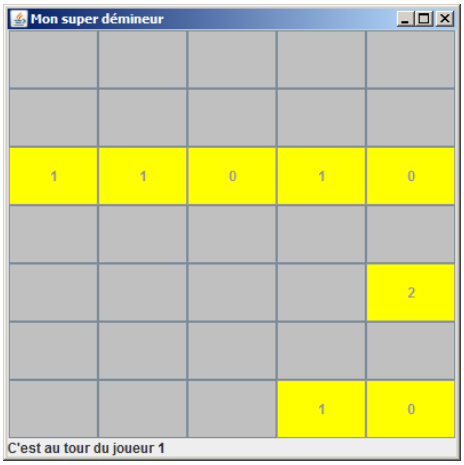
Au début du jeu, le ou les utilisateurs voient une aire de jeu sous la forme suivante :




Chaque case est en réalité un bouton (extension de JButton) qui représente des cases d'un démineur (classe `CaseDemineur`). Les cases peuvent être de type `Mine`, `Demine` ou `Info` comme le montre la simulation ci-dessous (les cases les plus sombres sont des Mines).

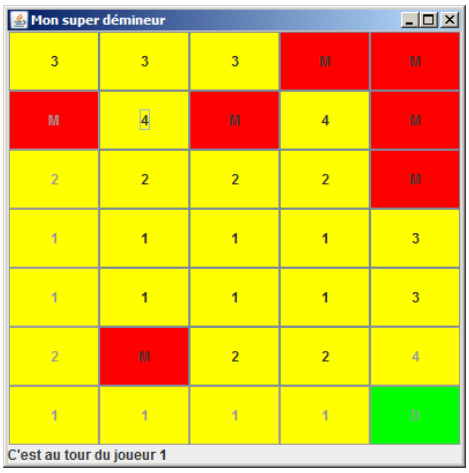


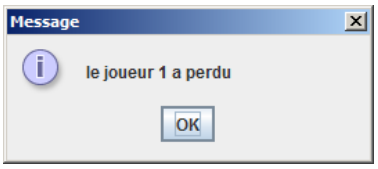

Un clic sur une case de type `Info` révèle le nombre de mines présentes sur la ligne et la colonne de la case sélectionnée.




Un clic sur une case de type `Demine` révèle le nombre de mines présentes sur la ligne de la case sélectionnée.

Un clic sur une case de type Mine révèle toute l'aire de jeu et fait perdre la partie à celui qui l'a activée.




Le code partiel de l'application est le suivant. Certaines parties seront à expliquer, réécrire, compléter.

```
public class Launcher {

	public static void main(String [] args){
		int nbJoueurs = 2;
		int nbLignes = 7;
		int nbColonnes = 5;

		Partie partie = new Partie(nbJoueurs, nbLignes, nbColonnes);
		JFrame frame = new DemineurGUI(nbJoueurs, nbLignes, nbColonnes, partie);
		// …
		frame.setVisible(true);
	}
}
public class DemineurGUI extends JFrame {

	private Partie partie;
	private JLabel labelInfo = new JLabel("C'est au tour du joueur 1");
	private Container grille ;

	public DemineurGUI(int nbJoueurs, int nbLigne, int nbCol, Partie partie){
		super("Mon super démineur");
		this.partie = partie;
		List<CaseDemineurIHM> listCaseDemineurIHM =
					this.partie.getListCaseDemineurIHM();
		initGrille(nbJoueurs, nbCol, nbLigne, listCaseDemineurIHM);
	}
	private void initGrille(int nbJoueurs, int nbLigne, int nbCol,
			List<CaseDemineurIHM> listCaseDemineurIHM) {

		this.grille = new JPanel();
		this.grille.setLayout(new GridLayout(nbCol, nbLigne));

		for(CaseDemineurIHM caseDemineurIHM : listCaseDemineurIHM){
			JButton bt = new BoutonDemineur(caseDemineurIHM);
			bt.addActionListener(new ListenerBt());
			grille.add(bt);
		}
		Container cont = getContentPane();
		cont.setLayout(new BorderLayout());
		cont.add(this.grille, BorderLayout.CENTER);
		cont.add(this.labelInfo,BorderLayout.SOUTH);
	}
```

**- Java 1 -**

```
	private class ListenerBt implements ActionListener {
		public void actionPerformed(ActionEvent ev) {
			BoutonDemineur source = (BoutonDemineur) ev.getSource();
			Resultat resultat = source.processAction();
			nextAction(resultat);
		}
	}
	private void nextAction(Resultat resultat) {
		this.repaint();

		if (Resultat.PERDU.equals(resultat) )
			JOptionPane.showMessageDialog(this,"le joueur " +
					partie.getNumJoueur() + " a perdu");
		else {
			if(partie.isEnd())
				JOptionPane.showMessageDialog(this,"le joueur " +
					partie.getNumJoueur() + " a gagné");
			else
				labelInfo.setText("C'est au tour du joueur " +
					partie.getNumJoueur());
		}
	}
}
public enum Resultat {
	ENCOURS,
	GAGNE,
	PERDU;
}
public class BoutonDemineur extends JButton {

	private CaseDemineurIHM caseDemineurIHM;
	private Color color = Color.LIGHT_GRAY;
	private String text = "";

	public BoutonDemineur(CaseDemineurIHM caseDemineurIHM) {
		super();
		this.caseDemineurIHM = caseDemineurIHM;
	}
	public void paintComponent(Graphics g){
		super.paintComponent(g);
		if (this.caseDemineurIHM.isVisible()) {
			this.color = this.caseDemineurIHM.getCouleur();
			this.text = this.caseDemineurIHM.toString();
		}
		this.setBackground(this.color);
		this.setText(this.text);
	}
	public Resultat processAction() {
		this.setEnabled(false);
		return this.caseDemineurIHM.processAction();
	}
}
public class CaseDemineurIHM {

	private CaseDemineur caseDemineur;

	public CaseDemineurIHM(CaseDemineur caseDemineur) {
		this.caseDemineur = caseDemineur;
	}
	public Color getCouleur() { return this.caseDemineur.getCouleur(); }
	public boolean isVisible() { return this.caseDemineur.isVisible(); }
	public String toString() { return this.caseDemineur.toString(); }
	public Resultat processAction() { return this.caseDemineur.processAction(); }
}
```

- COO 1 -

```
public class Partie {

	public static final int MAX_JOUEURS = 4;
	private Plateau plateau;
	private int nbJoueurs = 1;
	private int joueur = 1;

	public Partie(int nbJoueurs, int nbLignes, int nbColonnes){

		this.nbJoueurs = (nbJoueurs > 0 && nbJoueurs < Partie.MAX_JOUEURS)
				? nbJoueurs : 1;
		Map<CaseDemineur,Integer> mapFactory = new HashMap<CaseDemineur,Integer>();
		mapFactory.put(new Demine(), 5);
		mapFactory.put(new Mine(), 20);
		mapFactory.put(new Info(), 100);
		plateau = new Plateau(nbLignes, nbColonnes, mapFactory);
	}
	public Partie(int nbLignes, int nbColonnes){
		this(1, nbLignes, nbColonnes);
	}
	public int getNumJoueur() {
		return joueur;
	}
	private void nextJoueur(){
		joueur = (joueur%nbJoueurs) +1;
	}
	public boolean isEnd(){
		this.nextJoueur();
		return this.plateau.isEnd();
	}
	public List<CaseDemineurIHM> getListCaseDemineurIHM() {
		return plateau.getListCaseDemineurIHM();
	}
}
```

- Java 2 -

```
public class Plateau {

	private int nbLignes = 8;
	private int nbColonnes = 8;
	private List<List<CaseDemineur>> plateauDeCaseDemineur = null;
	private List<CaseDemineurIHM> listCaseDemineurIHM = null;
	private Map<CaseDemineur, Integer>  mapFactory = null;
	private int nbCaseMine = 0;

	public Plateau(int nbLignes, int nbColonnes, Map<CaseDemineur, Integer> mapFactory) {
		this.nbLignes = nbLignes;
		this.nbColonnes = nbColonnes;
		this.mapFactory = mapFactory;
		this.plateauDeCaseDemineur = this.initPlateauDeCaseDemineur();
		this.listCaseDemineurIHM = this.initListCaseDemineurIHM();
	}
	public List<CaseDemineurIHM> getListCaseDemineurIHM() {
		return this.listCaseDemineurIHM;
	}
	private List<CaseDemineurIHM> initListCaseDemineurIHM() {
		List<CaseDemineurIHM> listIHM = new LinkedList<CaseDemineurIHM>();
		CaseDemineurIHM caseDemineurIHM = null;
		for(List<CaseDemineur> uneLigneDeCases : this.plateauDeCaseDemineur){
			for(CaseDemineur uneCase : uneLigneDeCases){
				caseDemineurIHM = new CaseDemineurIHM(uneCase);
				listIHM.add(caseDemineurIHM);
			}
		}
		return listIHM;
	}
	public int getNbLignes() { return this.nbLignes; }
```

```
	public int getNbColonnes() { return this.nbColonnes; }

	private List<List<CaseDemineur>> initPlateauDeCaseDemineur() {   - ReDo -
		List<List<CaseDemineur>> plateauDemineur =
                                                    new ArrayList<List<CaseDemineur>>();
		CaseDemineur uneCaseDemineur;
		for (int i=0; i < this.nbLignes; i++){
			List<CaseDemineur> uneLigneDeCases = new ArrayList<CaseDemineur>();
			for(int j=0; j < this.nbColonnes; j++){
				uneCaseDemineur = this.tirage(j,i);
				uneLigneDeCases.add(uneCaseDemineur);
				if (uneCaseDemineur instanceof Mine) {
					nbCaseMine++;
				}
			}
			plateauDemineur.add(uneLigneDeCases);
		}
		return plateauDemineur;
	}
	private CaseDemineur tirage(int x, int y){
		CaseDemineur caseDemineur = null;
		int alea = (int)(Math.random()*100);
		boolean trouve = false;
		Iterator<Entry<CaseDemineur, Integer>> it =
			this.mapFactory.entrySet().iterator();

		while (!trouve && it.hasNext()) {
			Map.Entry<CaseDemineur, Integer> entry =
					(Map.Entry<CaseDemineur, Integer>)it.next();
			if(entry.getValue().intValue() > alea){
				caseDemineur = (CaseDemineur) entry.getKey().clone();
				caseDemineur.setPlateau(this);
				caseDemineur.setCoord(x, y);                          - Java 3 -
				trouve = true;
			}
		}
		return caseDemineur;
	}
	public void actionMine() {

		- ToDo 1 -
	}
	public void actionDemine(int y) {

		- ToDo 2 -
	}
	public int actionInfo(int x, int y) {

		- ToDo 3 -
	}
	public boolean isEnd() {

		- ToDo 4 -
	}
}

public abstract class CaseDemineur implements Cloneable{
	private int x, y;
	private boolean visible = false;
	private Color couleur;
	protected static Plateau plateau;                                 - Java 4 -

	public CaseDemineur(){
		this(0,0, Color.black);
	}
```

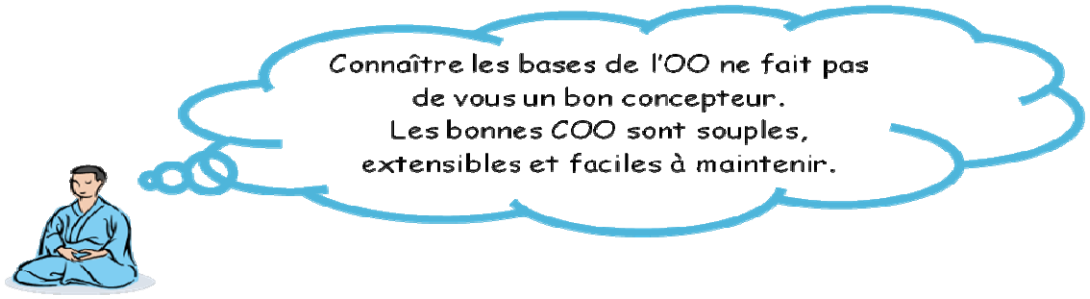
```
	public CaseDemineur(int x, int y, Color couleur){
		this.x = x;
		this.y = y;
		this.couleur = couleur;
	}
	protected int getX(){
		return this.x;
	}
	protected int getY(){
		return this.y;
	}
	public Color getCouleur() {
		return couleur;
	}
	public boolean isVisible() {
		return visible;
	}
	public void setVisible(boolean visible) {
		this.visible = visible;
	}
	public void setCoord(int x, int y){
		if(plateau.getNbLignes() > y && plateau.getNbColonnes() > x){
			this.x = x;
			this.y = y;
		}
	}
	public void setPlateau(Plateau plateau) {
		CaseDemineur.plateau = plateau;
	}
	public String toString(){
		String ret = "";
		if (this.visible) {
			ret = getClass().getSimpleName().substring(0, 1) ;
		}
		return ret;
	}
	public abstract Object clone();

	public abstract  Resultat processAction();
}
public class Info extends CaseDemineur {
	private int nbMines = 0;

	public Info(){ super();
	}
	public Info(int x, int y) { super(x, y, Color.YELLOW);
	}
	public Object clone() { return new Info(getX(),getY());
	}
	public void setVisible(boolean visible) {
		super.setVisible(visible);
		this.nbMines = plateau.actionInfo(this.getX(), this.getY());
	}
	public String toString(){
		String ret = "" + this.nbMines;
		if(!isVisible()){
			ret = super.toString();
		}
		return ret;
	}
	public Resultat processAction() {
		setVisible(true); // invoque plateau.actionInfo()
		return Resultat.ENCOURS;
	}
}
```

```
public class Mine extends CaseDemineur{
	public Mine(){ super();
	}
	public Mine(int x, int y) { super(x, y, Color.red);
	}
	public Object clone() { return new Mine(getX(),getY());
	}
	public Resultat processAction() {
		setVisible(true);
		plateau.actionMine();
		return Resultat.PERDU;
	}
}
public class Demine extends CaseDemineur {

	public Demine(){ super();
	}
	public Demine(int x, int y) { super(x, y, Color.green);
	}
	public Object clone(){ return new Demine(this.getX(),this.getY());
	}
	public Resultat processAction() {
		setVisible(true);
		plateau.actionDemine(this.getY());
		return Resultat.ENCOURS;
	}
}
```



Nom - Prénom :

## 1. Questions de compréhension de la conception du programme (4 + 2 points)

**- COO 0 -**

**Au brouillon, si cela vous aide, réalisez le diagramme de classe** (à vide) pour identifier les relations entre les classes. **Vous ne devrez pas rendre ce brouillon**. Comprenez bien l'utilité de la classe `CaseDemineurIHM` et observez bien comment elle est codée : elle sert d'interface entre le modèle et la vue. Le modèle, la classe `Partie`, est implémenté par un `Plateau` lui-même constitué de `CaseDemineur`. La vue est une `JFrame` composée d'1 `JPanel` et d'un `JLabel`. Le `JPanel` est composé de plusieurs `BoutonDemineur` qui enveloppent (contiennent) chacun 1 `CaseDemineurIHM`.

Je vous conseille de lire les questions COO 1 à 3 mais peut être de n'y répondre que lorsque vous vous serez bien approprié le code, c'est-à-dire après avoir répondu aux questions Java 1 à 4.

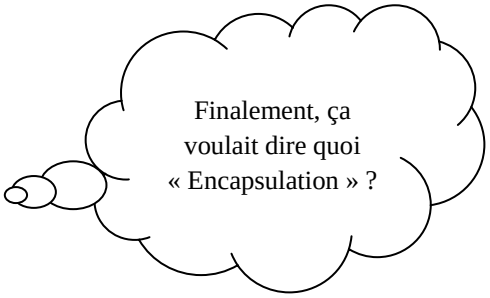
**- COO 1 -**

Quel est l'intérêt de la classe `Plateau`. En d'autres termes, aurait-on pu fusionner les classes `Partie` et `Plateau`? **Justifiez votre réponse en insistant sur les avantages/inconvénient de chaque approche.**

**- COO 2 -**

Une évolution du programme serait l'apparition de cases qui révèleraient les mines présentes dans leur colonne lorsqu'un joueur cliquerait dessus (en plus de celles qui révèlent les mines présentes sur leur ligne). **Expliquez quelles classes/méthodes seraient ajoutées/modifiées/supprimées** ?

**- COO 3 -**




Lorsqu'un joueur clique sur un bouton (`BoutonDemineur`), l'écouteur invoque sa méthode `processAction()` qui elle-même invoque la méthode `processAction()` de l'objet `CaseDemineurIHM` qu'il contient, qui elle-même invoque la méthode `processAction()` de l'objet `CaseDemineur` qu'il contient. Conceptuellement, la classe `CaseDemineurIHM` est ce que l'on appelle un adaptateur et elle adapte 1 objet instance de `CaseDemineur` pour changer son interface (dans le cas présent, elle propose moins de fonctionnalités). Pourquoi avoir ajouté ce niveau d'abstraction supplémentaire. N'aurait-il pas été possible de se passer de cet objet et que l'objet `Partie` expose directement à la vue (`DemineurGUI`) ses objets `CaseDemineur` (ceux de son `Plateau`) ? **Discutez des avantages et inconvénients de chacune de ces options.**

## 2. Questions de compréhension du code (6 points)

**- Java 1 -**

```
for(CaseDemineurIHM caseDemineurIHM : listCaseDemineurIHM){ … }
```

**Expliquez en français les actions effectuées par les instructions de cette boucle** ?

**- Java 2 -**

```
Map<CaseDemineur,Integer> mapFactory = new HashMap<CaseDemineur,Integer>();
mapFactory.put(new Demine(), 5); // …
```

La Map référencée par mapFactory a 2 objectifs :

- Elle permet à la classe Plateau de créer des CaseDemineur à partir d'autres CaseDemineur en utilisant la méthode clone() (la clé de chaque entrée dans la map est une instance de CaseDemineur),
- et pour chaque type de CaseDemineur, elle indique un nombre (5, 20, 100) qui sera utilisé par la fonction de tirage aléatoire (c'est la valeur de chaque entrée dans la map) : si le nombre généré aléatoirement est <5 alors un objet Demine est créé, s'il est >=5 et <20 alors, un objet Mine est créé, s'il est >=20 et <100 alors, un objet Info est créé.

**Aurait-on pu inverser la clé et la valeur, c'est-à-dire avoir une Map constituée de couples <Integer, CaseDemineur> et non pas <CaseDemineur, Integer>. Justifiez.**

**- Java 3 -**

```
caseDemineur.setPlateau(this);
caseDemineur.setCoord(x, y);
```

**Pourquoi faut-il utiliser ces « Setter » ? N'aurait-il pas été possible d'initialiser les attributs des objets CaseDemineur au moment de leur construction ? Expliquez.**

**- Java 4 -**

```
protected static Plateau plateau;
```

**Pourquoi cet attribut est-il déclaré avec le qualificateur static dans la classe CaseDemineur ? Expliquez.** Pour mémoire, un attribut static n'est pas spécifique à un objet instance d'une classe, il est partagé entre tous les objets instances de la même classe qui peuvent le lire et le modifier.

## 3. Maintenance évolutive du code (10 points)

**- ReDo –**

```
private List<List<CaseDemineur>> plateauDeCaseDemineur = initPlateauDeCaseDemineur()
private List<List<CaseDemineur>> initPlateauDeCaseDemineur() { … }
```

Vous vous demandez quelle idée a traversé l'esprit du développeur d'implémenter un tableau à 2 dimensions dans une `List<List<CaseDemineur>>` et vous proposez de redéfinir l'attribut `plateauDeCaseDemineur` ainsi : `private CaseDemineur plateauDeCaseDemineur[nbLignes][nbColonnes] ;`

**Réécrivez la méthode en conséquence :**

```
private CaseDemineur[nbLignes][nbColonnes] initPlateauDeCaseDemineur() {
```

```
}
```

Le chef de projet ne partage pas votre point de vue et vous demande de garder la déclaration initiale de l'attribut (`List<List<CaseDemineur>>`) et de restaurer le code antérieur de la méthode `initPlateauDeCaseDemineur()` avant de développer les méthodes manquantes (faites un reset dans votre tête, comme si cette question - ReDo - n'avait pas existé ☺).

**- ToDo 1à4 –**

**Complétez le code de toutes les méthodes suivantes selon les spécifications données en début de sujet et en étudiant bien le code des méthodes qui les invoquent** (par exemple, la méthode `processAction()` de la classe `Mine` invoque la méthode `actionMine()` de la classe `Plateau`).

```
public void actionMine() {









}
public void actionDemine(int y) {









}
public int actionInfo(int x, int y) {









}
```

```
public boolean isEnd() {









}
```

## Trucs et astuces

**Le TAD « Tables associatives »**

- Une table associative (interface Map) permet de conserver et de retrouver la valeur associée à une clé donnée.
- Exemples :
  - Dictionnaire : mot = clé, définition = valeur.
- Implémentations :
  - Sous forme de table de hachage : classe HashMap,
  - Sous forme d'arbre binaire : classe TreeMap.
- Parcours d'une Map : les Map n'implémentent pas l'interface Iterable et de ce fait on ne peut a priori pas les parcourir. Cependant, il existe une méthode entrySet() qui retourne une vue de la Map sous forme d'un ensemble (Set) qui lui peut être parcouru :

```
Map < Type clé, Type valeur > map ;
      // … Actions sur la Map …
Set < Map.Entry < Type clé, Type valeur > > set = map.entrySet();
for (Entry< Type clé, Type valeur > e : set){
      // manipulation de e.getKey()
      // manipulation de e.getValue()
}
```

Oubien

```
Iterator<Entry< Type clé, Type valeur >> it = set .iterator();
while (it.hasNext()) {
      Map.Entry< Type clé, Type valeur > entry = it.next();
      // manipulation de entry.getKey()
      // manipulation de entry.getValue()
}
```

- Ajout d'un élément dans la Map : `map.put(clé, valeur);`

**Le TAD « Liste »**

- Une liste (List) :
  - Admet des doublons, reconnait les indices.
  - Est munie d'une relation d'ordre, souvent l'ordre d'insertion ou l'ordre induit par un comparateur après un tri.
- Le TAD LISTE dont le comportement est défini par l'interface List est implémenté :
  - Sous forme de liste doublement chainée par la classe LinkedList.
  - Sous forme de vecteur dynamique (tableau) par les classes ArrayList et Vector (obsolète).
- Parcours : les List implémentent l'interface Iterable donc peuvent être parcourues à l'aide d'un itérateur (ou d'une boucle for-each qui masque l'utilisation de l'itérateur).

```
for (TypeElement e : maListe) { // manipulation de e }
```

- Ajout d'un élément en fin de liste : `boolean add(TypeElement e);`