Yann.Secq@univ-lille.fr
Fabien.Delecroix@univ-lille.fr

Objectifs: Utiliser des tableaux multi-dimension. S'habituer à la solution de problèmes en les décomposant.

1 Jeu de la vie

Exercice 1 : Jeu de la vie

Vous allez implémenter le programme `JeuDeLaVie.java` qui simule le jeu de la vie présenté en cours. On rappelle ici les fonctions à écrire, que vous êtes invités à faire dans cet ordre.

1. Écrire la fonction `void afficher(boolean[][] monde)` qui affiche le tableau `monde` représentant le monde du jeu de la vie. Vous choisirez un « grand » caractère pour les cases vivantes (i.e. les cases égales à `true`), par ex. `'@'` ou `'*'`, et un « petit » caractère pour les cases mortes, par ex. `'.'`.

   Rappelons que les cases sur la bordure du monde ne sont pas significatives et servent uniquement à faciliter les calculs, on peut donc ne pas les afficher.

   Pour tester la fonction d'affichage, écrire le `void algorithm()` qui initialise un tableau de taille 5×5 cases et l'affiche.

2. Écrire la fonction `void init(boolean[][] tab, double proba)` qui initialise le `tab` donnée en paramètre de manière à ce que chaque case ait une probabilité `proba` d'être vivante. On suppose que le tableau `tab` est déjà créé. Rappelons que toutes les cases sur les bordures doivent être mortes.

   Pour tester la fonction que vous venez d'écrire, modifiez `void algorithm()` en utilisant `init` pour initialiser le tableau, que vous affichez ensuite.

3. Écrire la fonction `int nombreDeVoisins(boolean[][] monde, int lig, int col)` qui compte le nombre de voisins vivants de la case aux coordonnées `lig, col` dans `monde`. Utilisez cette fonction de test pour tester votre fonction.

```
void testNombreDeVoisins() {
   // Initialiser un monde exemple
   boolean[][] monde =
       new boolean[][]{{false, false, false, false, false},
                       {false, false, false, false, false},
                       {false, true, true, true, false},
                       {false, false, false, false, false},
                       {false, false, false, false, false}};
   assertEquals(2, nombreDeVoisins(monde, 2, 2));
   assertEquals(2, nombreDeVoisins(monde, 1, 1));
   assertEquals(3, nombreDeVoisins(monde, 3, 2));
}
```

4. Écrire la fonction `boolean evolution(int nombreDeVoisins, boolean etatCellule)` qui calcule l'état d'une cellule à la génération suivante en fonction de son nombre de voisins et de son état actuel.

   La fonction doit passer ce test.

```
void testEvolution () {
   assertEquals(false,evolution(0, false));
   assertEquals(false,evolution(0, true));
   assertEquals(false,evolution(1, false));
   assertEquals(false,evolution(1, true));
   assertEquals(false,evolution(2, false));
   assertEquals(true,evolution(2, true));
   assertEquals(true,evolution(3, false));
   assertEquals(true,evolution(3, true));
   assertEquals(false,evolution(4, false));
   assertEquals(false,evolution(4, true));
}
```

5. Vous êtes maintenant prêt à écrire l'algorithme principal qui simulera le jeu de la vie, en complétant ce squelette:

```
void algorithm() {
   boolean [][] monde = new boolean[20][20];
   boolean [][] mondeN;
   int generation = 0;
   boolean stop = false;

   init(monde, 0.1);
   // Ici, on pourra par la suite ajouter des motifs (carré, glisseur, barre, autres)
=> voir question suivante
   afficher(monde);

   while(!stop){
      generation = generation +1;

      mondeN = new boolean[length(monde,1)][length(monde,2)];
      // Complétez ici par la double boucle qui calcule la
      // génération suivante dans mondeN
      ...

      monde = mondeN;

      println("Génération "+generation);
      afficher(monde);
      println("Entrez \"stop\" pour arrêter, ou entrée pour continuer");

      stop = equals("stop",readString());
   }
}
```

6. Vous allez maintenant écrire trois fonctions pour inclure des motifs, tester votre programme et mieux savourer le fruit de votre travail. Les fonctions en question vont générer comme motif :
   — un carré de taille 2×2 cases, dont on sait qu'il vivra éternellement (s'il n'est pas perturbé, bien entendu)
   — un « glisseur », qui en 4 générations effectue une translation en diagonale
   — une barre horizontale de 3 cases dont on sait qu'elle oscille en alternant avec une barre verticale.
   Chacune des trois fonctions prend en paramètre les coordonnées où (le coin supérieur gauche du) motif doit être ajouté. Écrire ces trois fonctions :

```
      // Ajoute ce motif aux coordonnées données
      // @@
      // @@
      void ajouterCarre (boolean[][] monde, int lig, int col)

      // Ajoute ce motif aux coordonnées données
      // .@.
      // ..@
      // @@@
      void ajouterGlisseur (boolean[][] monde, int lig, int col)

      // Ajoute ce motif aux coordonnées données
      // @@@
      void ajouterBarre (boolean[][] monde, int lig, int col)
```

2 Variations autour du démineur

Les deux exercices qui suivent sont des variations sur le jeu du démineur, où le joueur est devant un champ de « mines » dont les emplacements sont inconnus, et qu'il ne doit pas faire sauter. Ces exercices permettent de continuer à s'exercer aux tableaux à deux dimensions et à la décomposition des problèmes complexes en fonctions plus simples.

Exercice 2 : Qui sautera en dernier

Dans ce jeu la joueuse va ouvrir des cases du plateau jusqu'à ce qu'elle tombe sur une mine. C'est un jeu de hasard dont l'objectif est d'ouvrir le plus grand nombre de cases avant de tomber sur une mine.

Commencez par créer un fichier `Demineur.java`.

Nous avons besoin de deux tableaux à deux dimensions de booléens pour représenter l'état actuel du jeu:
— un champ qui contient des mines (true) ou des cases vides (false)
— une carte où chaque case peut être explorée (true) ou non explorée (false)

1. Écrire la fonction `void afficher(boolean[][] champ, boolean[][] carte)` qui affiche l'état actuel du jeu du point de vue de la joueuse, à savoir:
   — un point d'interrogation pour les cases non encore explorées
   — un point pour les cases explorées qui ne contiennent pas de mine
   — le symbole `'@'` pour les cases explorées qui contiennent une mine
   ce De plus, comme la joueuse devra saisir des coordonnées, il serait utile d'afficher les coordonnées des lignes et des colonnes en plus du contenu des cases. Voici ce à quoi l'affichage doit ressembler pour un champ de taille 3×5.

   ```
     ABCDE
   1 ??.?.
   2 .?..?
   3 ?..@?
   ```

   Pour tester la fonction, écrire `void algorithm()` qui initialise deux petits tableaux (2×2 cases suffisent) contenant les différentes possibilités, puis effectue l'affichage.

2. Écrire la fonction `void initialiserChamp(boolean[][] champ, double proba)` qui pose des mines dans les cases de `champ` avec la probabilité donnée en paramètre. Cette fonction est quasiment identique à celle écrite pour le jeu de la vie, sauf qu'ici les cases sur la bordure ne sont pas forcément vides.

   Tester la fonction grâce à `void algorithm()`, et utilisant l'affichage avec une carte totalement explorée.

   Notez qu'il n'est pas nécessaire d'écrire une fonction qui initialise la carte. Au départ la carte est entièrement non explorée (toutes les cases sont à `false`), et lors de l'allocation d'un tableau de booléens en java, toutes ses cases sont initialisées à `false`.

3. Écrire les deux fonctions de saisie des coordonnées. La coordonnée de la ligne est un nombre entre 1 et le nombre de lignes, et la coordonnée de la colonne est une lettre entre A et la lettre qui correspond au nombre de colonnes. Chacune de ces deux fonctions prend en paramètre le nombre de lignes (ou colonnes) et redemande la saisie tant que la joueuse saisit une coordonnée incorrecte.

   ```
   // Saisit une coordonnée de ligne correcte et retourne l'indice de ligne
   // qui correspond à cette coordonnée
   int saisirLigne (int nombreLignes)
   // Saisit une coordonnée de colonne correcte et retourne l'indice de colonne
   // qui correspond à cette coordonnée
   int saisirColonne (int nombreColonne)
   ```

   Comme ces fonctions font des saisies, il faut encore une fois les tester avec `void algorithm()`, par exemple en faisant des saisie et en affichant le résultat retourné par la fonction pour s'assurer qu'il est correct:

   ```
   void algorithm () {
      int l = saisirLigne(5);
      println(l); // doit afficher 3 si la joueuse a saisi 4

      int c = saisirColonne(10);
      println(c); // doit afficher 5 si la joueuse a saisi F
   }
   ```

   Une fois les tests effectués, il faut renommer la fonction `void algorithm()` en, par ex., `algorithm_testSaisie`

4. Vous êtes maintenant prêt à écrire l'algorithme principal, en complétant ce squelette.

   ```
   void algorithm () {
      boolean[][] champ = new boolean[5][7];
      boolean[][] carte = new boolean[5][7];

      boolean perdu = false;
      int score = 0;

      initialiserChamp(champ, 0.1);
      while (! perdu) {
         println("Score: " + score);
         afficher(champ, carte);
   ```

```
        println("Où veux-tu tenter ta chance ?");
        // Saisir les coordonnées et découvrir la case correspondante
        // sur la carte
        ...

        // Mettre à jour le score ou perdu
        ...
    }
    afficher(champ, carte);
    println("BOUM !");  // Dans ce jeu on perd toujours ...
    println("Ton score final est " + score);
}
```

Exercice 3 : Traverser un champ de mines

Dans ce jeu, l'objectif est de traverser le champ de mines sans marcher sur une mine. C'est donc un jeu où la joueuse peut gagner.

Commencez par créer le fichier `Demineur2.java`.

Au départ la joueuse se trouve sur la première colonne et la ligne du milieu. À chaque étape, elle doit choisir dans quelle direction aller parmi Haut (H), Bas (B) ou Avancer (A). Avancer consiste à se déplacer sur la colonne à droite. Notons qu'on ne peut pas revenir en arrière. La joueuse gagne si elle atteint la dernière colonne sans avoir marché sur une mine.

Suivre les indications pour écrire les fonctions nécessaires. Notez que certaines fonctions peuvent être directement reprises de l'exercice précédent.

L'état du jeu est représenté par le champ et la carte comme précédemment, mais aussi par les coordonnées où se trouve la joueuse actuellement.

1. Écrire la fonction `void afficher(boolean[][] champ, boolean[][] carte, int posJL, int posJC)` qui affiche le champ en indicant la position de la joueuse par `'&'`, les parties non explorées par `'?'`, les parties explorées sans mine par un espace, et la mine par `'@'`. Les paramètres `posJL, posJC` sont les indices de ligne/colonne où la joueuse se trouve actuellement. Voici en exemple : à gauche la configuration initiale, au centre la configuration après les déplacements A H H A, et à droite la configuration après les déplacements A H H A A B où la joueuse a marché sur une mine.

```
??????          ? &???          ?   ??
??????          ? ????          ? ?@??
&?????            ????            ????
??????          ??????          ??????
??????          ??????          ??????
```

   Tester la fonction en utilisant `void algorithm()`

2. Une commande est représentée par une des lettres parmi H, B, A. Écrire la fonction

```
boolean commandeEstValide(char commande, int posJL, int posJC,
                          int nbLignes, int nbColonnes)
```

   qui retourne vrai si la joueuse se trouvant à la position `posJL, posJC` peut effectuer `commande` sans sortir du champ, étant donné le nombre de lignes et de colonnes du champ. La fonction doit passer ce test

```
void testCommandeEstValide () {
    assertEquals(true,commandeEstValide('A', 2, 3, 5, 5));
    assertEquals(false,commandeEstValide('A', 2, 4, 5, 5));
    assertEquals(true,commandeEstValide('H', 1, 2, 5, 5));
    assertEquals(false,commandeEstValide('H', 0, 2, 5, 5));
    assertEquals(true,commandeEstValide('B', 3, 2, 5, 5));
    assertEquals(false,commandeEstValide('B', 4, 2, 5, 5));
}
```

3. Écrire la fonction
   `char saisirCommande(int posJL, int posJC, int nbLignes, int nbColonnes)`
   qui demande à la joueuse de saisir la commande à exécuter. Cette fonction va réitérer la saisie jusqu'à ce que la joueuse saisisse une commande valide qui peut être exécutée.

   La fonction écrite est à tester avec `void algorithm()`.

4. Écrire la fonction `int[] nouvellePosition(int posJL, int posJC, char commande)` qui retourne un tableau à deux cases contenant les nouvelles coordonnées de la joueuse après exécution de `commande`. La première case du résultat est le nouvel indice de ligne de la position, tandis que la deuxième case du résultat est le nouvel indice de colonne. On suppose que la commande donnée en paramètre est valide.

   (Optionnel) Écrire un test pour cette fonction.

5. Vous êtes maintenant prêt à écrire l'algorithme principal, en complétant ce squelette.

```
void algorithm () {
    boolean[][] champ = new boolean[5][7];
    boolean[][] carte = new boolean[5][7];
    boolean perdu = false;

    // Joueuse sur la ligne du milieu de la première colonne
    int posJL = length(carte,1)/2;
    int posJC = 0;

    initialiserChamp(champ, 0.1);
    // S'il y a une mine sur la position initiale, on l'enlève
    champ[posJL][posJC] = false;
    // La position initiale est explorée
    carte[posJL][posJC] = true;

    // Complétez. On s'arrête en cas de victoire ou en cas de défaite
    while (...) {
        afficher(champ, carte, posJL, posJC);

        // Saisir la commande et mettre à jour la carte et la position
        // de la joueuse
        ...

        // Mettre à jour la variable perdu
        ...
    }
    if (perdu) {
        println("BOUM ! ");
    } else {
        println("GAGNé!");
    }
    afficher(champ, carte, posJL, posJC);
}
```

3 Prolongements

Exercice 4 : Exercice de style: parcourir un tableau à deux dimensions avec une seule boucle

Revenons sur l'exercice qui recherche l'indice de la première occurrence d'un caractère dans un tableau de caractères à deux dimensions. La fonction doit passer ce test:

```
void testRechercherPremiereOccurrence () {
    char[][] tab = new char[][]{{'a','b','c'},{'d','a','b'},{'a','e','d'}};
    assertArrayEquals(new int[]{0,1}, rechercherPremiereOccurrence(tab, 'b'));
    assertArrayEquals(new int[]{1,0}, rechercherPremiereOccurrence(tab, 'd'));
    assertArrayEquals(new int[]{2,1}, rechercherPremiereOccurrence(tab, 'e'));
    assertArrayEquals(new int[]{-1,-1}, rechercherPremiereOccurrence(tab, 'x'));
}
```

Écrire cette fonction dans un programme `ParcoursTaleau` en n'utilisant qu'une seule boucle ! Indice: si `tab` est un tableau de taille 3×4, comment peut-on représenter par un entier tout couple (i, j) d'indices corrects de `tab`? Comment calculer i et j à partir de cet entier ?

Discuter de l'avantage de l'utilisation de la version à une seule boucle, en comparant avec la solution avec deux boucles imbriquées.