Université Denis Diderot - DEUG MIAS/MASS - IF121
Corrigé de l'examen du 17/01/2005

Exercice 1.

1.1. *Fonction testant l'alignement de trois points*

```
static boolean alignes(int xA, int yA, int xB, int yB, int xC, int yC) {
    int xAB, yAB, xAC, yAC;  // les coordonnées des vecteurs AB et AC
    xAB = xB - xA; yAB = yB - yA;  // calcul de celles de AB
    xAC = xC - xA; yAC = yC - yA;  // calcul de celles de AC
    return xAB * yAC == yAB * xAC; // retour de la valeur du test d'égalité
}
```

1.2. *Fonction calculant le nombre de points alignés avec deux points donnés*

La fonction appelle le nombre de fois nécessaire la fonction précédente et incrémente un entier initialisé à 0 chaque fois qu'un point aligné avec les deux premiers est trouvé.

```
static int alignes2(int[ ] x, int[ ] y) {
   if (x.length != y.length || x.length < 3) // test de correction des paramètres
        return -1;    // retour -1 si paramètres incorrects
   int nombre = 0;    // nombre courant de points alignés avec les deux premiers
   int i = 2;         // prochain point à tester
   while (i < x.length) {  // travail fini ?
      if(alignes(x[0], y[0], x[1], y[1], x[i], y[i]))
          nombre++;  // si le point est aligné; incrémenter nombre
      i++;           // passer au point suivant
   }
   return nombre;    // renvoyer le nombre de points alignés
}
```

1.3. La représentation des nombres réels en machine sous forme binaire et en virgule flottante fait que les nombres sont représentés sous forme approchée. De plus des arrondis sont réalisés. Cela fait que le test d'égalité de deux valeurs égales sous leur forme décimale peut conduire à une réponse négative. Les comparaisons des nombres doivent donc être réalisés à ϵ près.

Exercice 2.

2.1. *Diviseurs stricts d'un entier $n > 0$*

2.1.1. Une borne supérieure du plus grand diviseur strict d'un entier $n > 0$ est évidemment $n \div 2$, quotient de n par 2 dans la division euclidienne.

2.1.2. *Algorithme caclulant tous les diviseurs stricts d'un entier $n > 0$ donné*

- Il suffit de calculer les restes de toutes divisions de n par tous les entiers tels que $i \leq n \div 2$. Les diviseurs de n sont ceux dont ce reste est nul.
- Cela peut se traduire de manière plus fine en :

pour $i = 0$ jusqu'à $n \div 2$ faire
si le reste de la division entière de n par i est nul
alors i est un diviseur de n
sinon i n'est pas un diviseur de n

2.1.3. *Fonction Java renvoyant le tableau des diviseurs stricts de $n > 0$*

```
static int[ ] diviseurs(int n) {
   if(n <= 0) return null;  // si n est négatif ou nul, on renvoie null
   int m = n / 2;           // borne supérieure du + grand diviseur
   int[ ] t1 = new int[m];  // allocation d'un tableau suffisamment grand
   int nombre = 0;          // nombre de diviseurs déjà trouvés
   for (int i = 1; i <= m; i++)  // pour les entiers 1, 2, ..., m
      if(n % i == 0){  // test de reste nul
          t1[nombre] = i; // si reste nul, mémoriser i comme diviseur
          nombre ++;      // et incrémenter le nombre de diviseurs trouvés
      }
   if (nombre == m) return t1   // t1 est de bonne taille (n a m diviseurs)
   int[ ] t = new int[nombre];  // allouer un tableau de bonne taille
   // sinon, il faut ajuster le tableau à la bonne taille
   for (int i = 0; i <nombre; i++)
       t[i] = t1[i]; // recopie de la partie utile de t1 dans t
   return t;         // retour du tableau t de de bone taille
}
```

2.1.4. *Fonction Java renvoyant la somme des diviseurs stricts de $n > 0$*
La fonction appelle tout d'abord la fonction précédente, puis calcule la somme des éléments du tableau que cette appel a renvoyé :

```
static int sommeDiviseurs(int n){
   if (n <= 0) return -1;  // test que le paramètre est correct
   int[ ] t = diviseurs(n); // récupération du tableau des diviseurs stricts de n
   int somme = 0;          // somme des diviseurs déjà calculée
   for (int i = 0; i < t.length; i++) // parcours du tableau
       somme = somme + t[i];  // ajouter l'élément à la somme
   return somme;  // renvoyer la valeur
}
```

2.2. *Nombres amis*

2.2.1. *Fonction Java testant si deux entiers a et b sont ou non amis*
La fonction utilise évidemment la fonction précédente :

```
static boolean sontAmis(int a, int b) {
   if (a <= 0 || b <= 0) return false;  // test de correction des paramètres
   // comparaison de chaque nombre à la somme des diviseurs de l'autres
   return(sommeDiviseurs(a) == b && sommeDiviseurs(b) == a);
}
```

2.2.2. *Fonction `main` utilisant la fonction précédente*

```
public static void main(String[ ] st){
   int m = Deug.readInt();  // lecture d'un premier nombre
   int n = Deug.readInt();  // lecture d'un second nombre
   if (sontAmis(m, n))      // tester s'ils sont ou non amis
       Deug.println(m + " et " + n + " sont amis");
   else
       Deug.println(m + " et " + n + " ne sont pas amis");
}
```

Exercice 3.

Il n'était demandé que le code de la fonction appelée ici `nombreValeurs`, mais la solution donnée inclut aussi celui d'une fonction `main` et des exemples d'exécution :

```
--> cat ExamC.java
import fr.jussieu.script.*;
class ExamC {
   static int[ ] nombreValeurs(int[ ][ ] t, int n){
       int[ ] tt = new int[n+1];
       for(int i = 0; i <= n; i++) tt[i] = 0;
       for(int i = 0; i < t.length; i++)
           for(int j = 0; j < t[i].length; j++){
              if(t[i][j] < 0 || t[i][j] > n)
                   return null;
              tt[t[i][j]] ++;
            }
       return tt;
   }
   public static void main(String[ ] st) {
       int[ ][ ] t = {{1,2,3},
                      {0, 2, 3, 4},
                      {1, 3, 2 , 1 , 3, 2},
                      {0, 0 , 1 ,2}};
       int[ ] tt;
       int n = Deug.readInt();    // lecture de la valeur de n
       tt = nombreValeurs(t, n);  // appel de la fonction nombreValeurs
       if(tt == null)     // si la fonction renvoie null, il y a eu erreur
           Deug.println("Erreur");
       else {    // sinon
           for(int i = 0; i <= n; i++)  // pour chacun des éléments du tableau
               Deug.print(tt[i] + " "); // afficher sa valeur
           Deug.println();
           }
   }
}
--> java ExamC
3            <-- nombre saisi au clavier
Erreur       <-- le tableau contient 4 qui est > 3
--> java ExamC
4
3 4 5 4 1
-->
```

Exercice 4.

La solution a été décomposée en trois fonctions, chacune étant dédiée à un type de pièce donnée, en supposant a priori que les deux positions sont sur l'échiquier. La quatrième fonction vérifie que les deux positions sont sur l'échiquier et, si elles le sont, sélectionne la fonction appelée selon la pièce concernée.

- Un mouvement d'une tour est légal si les deux positions ont même abscisse et des ordonnées différentes ou ont même ordonnée et des abscisses différentes :

```
static boolean tour(int x1, int y1, int x2, int y2) {
    if(x1 == x2)    // positions sur la même colonne
        return (y1 != y2); // les lignes doivent être différentes
    if(y1 == y2)    // positions sur la même ligne
        return (x1 != x2); // les colonnes doivent être différentes
    return false;  // colonnes et lignes différentes : impossible
}
```

- Un mouvement d'un fou est légal si il conduit à un point différent sur sa diagonale ($x1 - x2 = y1 - y2$) ou sur son anti-diagonale ($x1 - x2 = y2 - y1$) :

```
static boolean fou(int x1, int y1, int x2, int y2) {
    if(x1==x2 && y1 == y2) return false;  // même position
     return ((x1-x2 == y1-y2)       // sur la diagonale
              || (x1-x2 == y2-y1)); // ou sur l'anti-diagonale
}
```

- Pour vérifier qu'un mouvement d'un cavalier est légal, le plus simple est de vérifer que les deux positions constituent les extrémités d'un triangle rectangle de petits côtés de longueur 1 et 2 : le carré de son hypoténuse 5 ou le produit du carré de ses côtés est égal à 4, d'où :

```
static boolean cavalier(int x1, int y1, int x2, int y2) {
    int x = x1 - x2; int y = y1 - y2;
    return ((x*x + y*y) == 5);   // ou encore : return x*x*y*y == 4;
}
```

On peut aussi écrire (`abs` calculant la valeur absolue) :

```
   return (   (abs(x1 - x2) == 2 && abs(y1 - y2) == 1)
           || (abs(x1 - x2) == 1 && abs(y1 - y2) == 2)));
```

La dernière fonction vérifie que les positions appartiennent à l'échiquier et réalise l'appel à la fonction correspondant à la pièce (erreur signalée si le paramètre de type `char` n'est pas un de ceux identifiant les pièces qui nous intéressent) :

```
static boolean mouvementPiece(int x1, int y1, int x2, int y2, char piece) {
     if(x1<0 || x1>7 || y1<0 || y1>7 || x2<0 || x2>7 || y2<0 || y2>7)
          return false;  // hors de l'échiquier
     switch(piece) {  // aiguillage selon la pièce
        case 'F' : // la pièce est un fou
                   return fou(x1, y1, x2, y2);
        case 'T' : // la pièce est une tour
                   return tour(x1, y1, x2, y2);
        case 'C' : // la pièce est un cavalier
                   return cavalier(x1, y1, x2, y2);
        default : // la pièce n'est pas correcte
                   return false;
     }
}
```

Exercice 5.

5.1. *Code de la classe* `ChequeAuPorteur`

La classe encapsule un nombre entier et possède un constructeur :

```
class ChequeAuPorteur {
   int montant;   // le montant du chèque
   ChequeAuPorteur(int valeur){montant = valeur;}  // constructeur
}
```

5.2.1. *Code de la classe* `Compte`

La classe encapsule deux entiers et un booléen initialisé à 0. Elle conient trois constructeurs et les différentes méthodes d'instance (non spécifiées `static`) demandées.

```
import fr.jussieu.script.*;
class Compte{
   private int solde, decouvert;
   boolean bloque = false;
   Compte(int depot){solde = depot; decouvert = 0;}
   Compte(int depot, int decouvert){
         solde = depot; this.decouvert = decouvert;}
   Compte(ChequeAuPorteur cheque, int decouvert){
         solde = cheque.montant; this.decouvert = decouvert;}
   int getSolde() { return solde;}
   int depot(int montant){
       solde += montant;
       if(bloque && solde > 0) bloque = false;
       return solde;
   }
   int retrait(int montant){
       if(bloque || solde-montant < -decouvert) return 0;
       solde -= montant;
       return montant;
   }
   int retraitDiffere(int montant){
       solde -= montant;
       if(solde < -decouvert) bloque = true;
       return solde;
   }
}
```

5.2.2. *Application de test de la classe Compte*

```
--> cat UtiliseCompte.java
class UtiliseCompte {
   public static void main(String[ ] st) {
       Compte c1 = new Compte(1000);
       Compte c2 = new Compte(2000, 500);
       ChequeAuPorteur cheque = new ChequeAuPorteur(1200);
       Compte c3 = new Compte(cheque, 150);
       int n = c1.retrait(350);
       Deug.println("retrait sur c1 de " + n + " --> Nouveau solde : " +
                        c1.getSolde());
       n = c2.depot(300);
```

```
        Deug.println("solde de c2 --> " + n);
        n = c3.retrait(1250);
        Deug.println("retrait sur c3 de " + n + " --> Nouveau solde : " +
                        c3.getSolde());
        n = c3.retrait(150);
        Deug.println("retrait sur c3 de " + n + " --> Nouveau solde : " +
                        c3.getSolde());
        n = c3.retraitDiffere(150);
        Deug.println("nouveau solde de c3 --> " + n);
        n = c3.depot(170);
        Deug.println("solde de c3 --> " + n);
        n = c3.retrait(10);
        Deug.println("retrait sur c3 de " + n + " --> Nouveau solde : " +
                        c3.getSolde());
        n = c3.depot(60);
        Deug.println("solde de c3 --> " + n);
        n = c3.retrait(100);
        Deug.println("retrait sur c3 de " + n + " --> Nouveau solde : " +
                        c3.getSolde());
    }
}
--> java UtiliseCompte
retrait sur c1 de 350 --> Nouveau solde : 650
solde de c2 --> 2300
retrait sur c3 de 1250 --> Nouveau solde : -50
retrait sur c3 de 0 --> Nouveau solde : -50
nouveau solde de c3 --> -200
solde de c3 --> -30
retrait sur c3 de 0 --> Nouveau solde : -30
solde de c3 --> 30
retrait sur c3 de 100 --> Nouveau solde : -70
-->
```