Programmation orientée objets : partiel
Énoncé 1

Pour les exercices d'affichage, vous devez impérativement répondre sur l'énoncé et le rendre en fin d'examen (dans une copie cachetée permettant de vous identifier après correction). Exceptée cette partie cachetée, rien ne doit pouvoir identifier votre copie et votre énoncé.

Affichage (1)

Affichage produit

```
public class Exo11 {

  public static void main(String[] args) {
    int x = 9;
    int j = -3;
    int k = 0;
    do {
      int z = j;
      x--;
      System.out.printf("%d+", j);
      if (k % 2 == 1) {
        j++;
        System.out.printf("%d+", k);
        z += k;
      }
      k++;
      System.out.printf("%d=%d%n", x, z + x);
    } while (x > 3 && j <= 2);
    System.out.printf("%d %d %d%n", x, j, k);
  }

}
```

Affichage (2)

Affichage produit

```
import java.util.Arrays;

public class Exo21 {

  public static void main(String[] args) {
    int[] x = new int[2];
    System.out.println(Arrays.toString(x));
    int[][] Y = new int[2][];
    Y[0] = x;
    Y[1] = new int[] { -1, 2, 3 };
    x = new int[] { -1, 2, 3 };
    for (int[] u : Y) {
      System.out.printf("%d : %d%n", u.length, u[0]);
      u[u.length - 1] = u.length + 3;
    }
    System.out.println(Arrays.deepToString(Y));
    System.out.println(Arrays.toString(x));
  }

}
```

Affichage (3)

```
import java.math.BigInteger;

public class Exo31 {

  public static void main(String[] args) {
    BigInteger x = BigInteger.ONE;
    BigInteger y = new BigInteger("20");
    BigInteger z = x;
    while (x.compareTo(y) < 0) {
      System.out.println(x);
      BigInteger w = x.remainder(BigInteger.valueOf(2));
      if (w.compareTo(BigInteger.ZERO) == 0) {
        z = z.add(BigInteger.valueOf(3));
      } else {
        z = z.add(BigInteger.valueOf(-1));
      }
      System.out.println(z);
      x = x.add(z);
    }
    System.out.printf("%s %s%n", x, z);
  }
}
```

Affichage produit

Programmation (1)

Pour tous entiers strictement positifs n et k, on définit Γ_n^k par récurrence en posant

$$\Gamma_1^k = 1,\ \forall k \geq 1,$$
$$\Gamma_n^1 = n,\ \forall n \geq 1,$$
$$\Gamma_n^k = \Gamma_n^{k-1} + \Gamma_{n-1}^k,\ \forall n > 1, k > 1.$$

Écrire un programme qui réalise les opérations suivantes :

1. demande à l'utilisateur deux valeurs entières P et Q en s'assurant qu'elles sont bien strictement positives ;
2. calcule les valeurs des Γ_n^k pour tout $n \leq P$ et tout $k \leq Q$;
3. affiche ces valeurs dans l'ordre croissant des n, et pour chaque n, dans l'ordre croissant des k (sur une seule ligne par valeur de n). Par exemple, pour $P = 2$ et $Q = 2$, on affichera sur une première ligne Γ_1^1 et Γ_1^2, puis sur une seconde ligne Γ_2^1 et Γ_2^2.

Il est vivement conseillé d'utiliser un tableau à deux dimensions pour réaliser ce programme.

Programmation (2)

Écrire un programme simple qui réalise exactement les mêmes calculs et affichages que le programme ci-dessus, mais en utilisant des `long` à la place des `BigInteger`. On s'efforcera de simplifier le programme par rapport à la version originale, quand cela est possible.

Programmation (3)

Pour calculer efficacement le reste de la division par m de b^e (pour tous entiers e, b et m strictement positifs), soit $b^e \mod m$, on utilise un algorithme dit d'exponentiation modulaire rapide (ce calcul est utile en cryptographie et doit être réalisé rapidement pour des nombres très grands). En pratique ceci consiste à calculer simultanément les suites $(r_n)_{n\geq 1}$, $(s_n)_{n\geq 1}$ et $(t_n)_{n\geq 1}$ données par :

$$r_1 = 1 \qquad s_1 = e \qquad t_1 = b$$
$$r_n = \begin{cases} r_{n-1}t_{n-1} \mod m & \text{si } s_{n-1} \text{ est impair} \\ r_{n-1} & \text{sinon} \end{cases} \qquad s_n = \frac{s_{n-1}}{2} \qquad t_n = (t_{n-1})^2 \mod m$$

Attention, dans ces équations $\frac{s}{2}$ désigne le *quotient* de la division euclidienne de s par 2, et $z \mod m$ désigne le reste de la division euclidienne de z par m. On remarque notamment que s_n est strictement décroissante : en pratique, le résultat du calcul est la valeur de r_k pour le premier k tel que $s_k = 0$.

Écrire un programme qui réalise les opérations suivantes (attention, *tous* les calculs doivent être effectués avec des `BigInteger` faute de quoi l'exercice sera considéré comme *entièrement* faux) :

1. demande à l'utilisateur les valeurs de trois `BigInteger` m, b et e (on pourra utiliser la méthode `nextBigInteger` de la classe `Scanner`) ;
2. calcule les suites $(r_n)_{n\geq 1}$, $(s_n)_{n\geq 1}$ et $(t_n)_{n\geq 1}$ en s'arrêtant dès qu'on atteint un k tel que $s_k = 0$;
3. affiche la valeur de r_k ;
4. affiche la valeur de b`.modPow(`e,m`)` pour valider les calculs (cette méthode effectue directement le calcul de $b^e \mod m$).

Affichage (4)

```
public class Exo41 {
  private StringBuilder sb;

  public Exo41() {
    sb = new StringBuilder();
  }
  public Exo41(int val) {
    this();
    sb.append(val);
  }

  public String toString() {
    return sb.toString();
  }

  public Exo41 a(Exo41 that) {
    Exo41 res = new Exo41();
    res.sb.append('(');
    res.sb.append(sb);
    res.sb.append(")%(");
    res.sb.append(that.sb);
    res.sb.append(")");
    return res;
  }

}

public class TestExo41 {

  public static void main(String[] args) {
    Exo41 x = new Exo41(10);
    Exo41 y = new Exo41(3);
    System.out.println(x);
    Exo41 z = y.a(x);
    System.out.printf("%s %s%n", y, z);
    z = z.a(new Exo41(-1));
    Exo41 w = z;
    z = z.a(new Exo41(5));
    System.out.printf("%s %s%n",z,w);
  }

}
```

Affichage produit

Programmation (4)

1. Proposer une version modifiable de la classe `Exo41` ci-dessus.
2. Réécrire la méthode `main` ci-dessus en utilisant la version modifiable de la classe et en faisant en sorte que les affichages obtenus soient exactement les mêmes qu'avec la version immuable.

Affichage (5)

Affichage produit

```
public class Exo51 {
  private boolean[] x;

  public Exo51(int n) {
    x = new boolean[n];
  }

  public boolean f(Exo61 y) {
    boolean res = false;
    if (y.getY() >= 0 && y.getY() < x.length) {
      res = x[y.getY()] != y.isT();
      x[y.getY()] = y.isT();
    }
    return res;
  }

  public String toString() {
    StringBuilder res = new StringBuilder();
    for (boolean t : x) {
      if (t) {
        res.append('X');
      } else {
        res.append('O');
      }
    }
    return res.toString();
  }
}

public class Exo61 {
  private int y;

  private boolean t;

  public Exo61(int y, boolean t) {
    this.y = y;
    this.t = t;
  }

  public int getY() {
    return y;
  }

  public boolean isT() {
    return t;
  }

}

public class TestExo56 {

  public static void main(String[] args) {
    Exo51 bla = new Exo51(5);
    System.out.printf("%s%n", bla);
    Exo61 foo = new Exo61(1, true);
    System.out.println(bla.f(foo));
    System.out.printf("%s %s%n", bla, foo);
    for (int i = 1; i < 6; i++) {
      foo = new Exo61(i, true);
      System.out.print(bla.f(foo));
      System.out.printf(" %s%n", bla);
    }
  }

}
```

Programmation (5)

On souhaite écrire des classes pour un jeu vidéo dans lequel des personnages s'affrontent. Les combats sont réglés en partie au hasard grâce à des lancers de dés.

Dés

On suppose donnée une classe `Dice` qui représente un groupe de dés (vous ne devez pas programmer cette classe). Plus précisément la classe possède les méthodes et constructeur suivants :

- le constructeur `Dice` prend deux paramètres : le nombre de dés du groupe et le nombre de faces des dés. Par exemple `Dice d = new Dice(3,4)` crée un objet `Dice` représentant un groupe de 3 dés comportant chacun 4 faces ;
- la méthode `roll` effectue un lancer des dés du groupe et renvoie un tableau de type `int[]` contenant les résultats du lancer. Par exemple, avec la variable `d` initialisée comme au dessus, on peut obtenir pour `d.roll()` un tableau `{2, 1, 2}` ;
- la méthode `rollSum` effectue un lancer des dés du groupe et renvoie la somme des valeurs obtenues. Dans l'exemple ci-dessus, `d.rollSum()` peut renvoyer une valeur comprise entre 3 et 12 ;
- enfin, la méthode `toString` renvoie une représentation du groupe de dés sous la forme d'une chaîne constituée du nombre de dés, suivi de la lettre D (en majuscule) suivie du nombre de faces. Pour le groupe utilisé en exemple, on obtient ainsi la chaîne `"3D4"`.

Écrire un programme qui utilise la classe `Dice` pour réaliser les opérations suivantes :

1. créer deux groupes de dés, A et B : A comporte 2 dés à huit faces et B comporte 3 dés à six faces ;
2. afficher une description des deux groupes ;
3. lancer 100 fois chaque groupe et afficher le nombre de fois que le groupe A a donné une somme supérieure strictement à celle du groupe B ;
4. lancer 100 fois chaque groupe et afficher le nombre de fois que la plus grande valeur obtenue par les trois dés du groupe B a été plus petite ou égale à la plus grande valeur obtenue par les deux dés du groupe A.

Armes

Chaque personnage dispose d'une arme décrite par un objet de type `Arme`. La classe `Arme` est de la forme suivante (le constructeur et les méthodes sont à compléter) :

```
public class Arme {
    private String name;

    private Dice damages;

    public Arme(String name, Dice damages) {    }

    public String toString() {    }

    public int hit() {    }
}
```

Compléter la classe en utilisant les indications suivantes :

- la variable d'instance `name` est le nom de l'arme ;
- la variable d'instance `damages` est le groupe de dés qui représente la létalité de l'arme (les dégâts qu'elle réalise) ;
- la méthode `toString` doit renvoyer une chaîne décrivant l'arme. Par exemple si on crée l'arme `a=new Arme("Dague",d)` où `d` désigne le groupe de dés créé par `d=new Dice(1,3)`, on s'attend à obtenir une chaîne de la forme `"Dague (1D3)"` ;
- la méthode `hit` renvoie les dommages effectivement causés par l'arme si elle touche un personnage : il s'agit de la somme des dés du groupe de dés représenté par `damages`. Concrètement dans l'exemple ci-dessus, ce sera donc une valeur entre 1 et 3 correspondant au lancer d'un unique dé à 3 faces. Attention, chaque appel à `hit` doit correspondre à un nouveau lancer du groupe de dés.

Personnages

Un personnage est représenté par un objet de la classe `Personnage` ci-dessous :

```
public class Personnage {
    private String name;

    private Dice dexterity;

    private int hitPoints;

    private Arme weapon;

    public Personnage(String name, Dice dexterity, int hitPoints) {    }

    public int getHitPoints() {    }

    public boolean isUnconscious() {    }

    public void equip(Arme weapon) {    }

    public String toString() {    }

    private boolean tryToHit() {    }

    public boolean takeDamage(int val) {    }

    public boolean attack(Personnage that) {    }
}
```

L'objectif de cette partie est de compléter la classe en programmant son constructeur et ses méthodes selon les indications ci-dessous. On utilisera aussi le programme qui suit les indications et donne un exemple d'utilisation de la classe. Il est vivement conseillé de lire l'intégralité des indications et l'exemple avant de compléter la classe.

- On note tout d'abord qu'un personnage possède donc quatre caractéristiques :
  1. son nom (`name`) ;
  2. sa dextérité (`dexterity`) ;
  3. ses points de vie (`hitPoints`) ;
  4. son arme éventuelle (`weapon`).
- La méthode `getHitPoints` renvoie la valeur actuelle des points de vie du personnage.
- Quand un personnage est créé, il ne possède pas d'arme (ce qui est représenté par la valeur `null`). Pour équiper un personnage, on doit appeler sa méthode `equip`.
- On s'attend à ce que la méthode `toString` tienne compte de l'équipement du personnage. Par exemple, le programme suivant

  ```
  Personnage gudrun = new Personnage("Gudrun", new Dice(1, 6), 100);
  System.out.println(gudrun);
  gudrun.equip(new Arme("Grande épée", new Dice(1, 10)));
  System.out.println(gudrun);
  ```

  devra afficher

  ```
  Gudrun (1D6, 100)
  Gudrun (1D6, 100) équipé de Grande épée (1D10)
  ```

  Après le nom du personnage, on indique donc sa dextérité puis ses points de vie, et enfin son arme éventuelle.
- Lors d'un combat, les points de vie baissent en fonction des coups subis (cf la méthode `takeDamage`). Si `hitPoints` devient négatif ou nul, le personnage est inconscient. La méthode `isUnconscious` doit alors renvoyer `true` (et `false` dans le cas contraire).
- La méthode `takeDamage` réalise les effets d'un coup : les points de vie baissent de la valeur `val` donnée en paramètre.
- Pour tenter de toucher un adversaire, on utilisera en interne (d'où le `private`) la méthode `tryToHit`. Cette méthode renvoie `true` si la tentative réussit (le personnage touche son adversaire) et `false` sinon. Pour déterminer si on touche, on effectue un lancer des dés du groupe `dexterity`. Si au moins un des dés donne 5 ou plus, la tentative est réussie.

— Enfin, pour attaquer un autre personnage, on utilise la méthode `attack`. L'attaque se décompose en deux étapes. On détermine d'abord si on touche (méthode `tryToHit` ci-dessus). Si c'est le cas, on inflige des dommages au personnage attaqué (`that`), en s'appuyant notamment sur la méthode `hit` de l'arme pour déterminer leur ampleur. Si on ne possède pas d'arme ou si on est inconscient, on ne peut pas attaquer. La méthode `attack` renvoie `true` en cas d'attaque réussie et `false` sinon.

Voici un exemple de combat entre deux personnages :

```
Personnage siegfried = new Personnage("Siegfried", new Dice(2, 6), 100);
siegfried.equip(new Arme("Grande épée", new Dice(1, 10)));
Personnage brunehilde = new Personnage("Brunehilde", new Dice(3, 6), 80);
brunehilde.equip(new Arme("Lance", new Dice(1, 6)));
System.out.printf("%s <-> %s%n", brunehilde, siegfried);
brunehilde.attack(siegfried);
siegfried.attack(brunehilde);
System.out.printf("%s <-> %s%n", brunehilde, siegfried);
```

On crée un premier personnage et on l'équipe d'une arme. On crée un second personnage, qu'on équipe aussi. Enfin, chaque personnage attaque l'autre. Le programme affiche par exemple :

```
Brunehilde (3D6, 80) équipé de Lance (1D6) <-> Siegfried (2D6, 100) équipé de Grande épée (1D10)
Brunehilde (3D6, 74) équipé de Lance (1D6) <-> Siegfried (2D6, 95) équipé de Grande épée (1D10)
```

On constate ici que chaque attaque a réussi puisque les points de vie des deux personnages ont diminué.

Combat

On suppose données deux variables `A` et `B`, de type `Personnage`, désignant chacune un personnage équipé d'une arme. On a donc un début de programme de la forme suivante :

```
Personnage A = ...;
A.equip(...);
Personnage B = ...;
B.equip(...);
```

L'objectif est de compléter le programme pour qu'il mettre en œuvre un combat entre les deux personnages. À chaque tour du combat, on choisit au hasard qui attaque en premier (une chance sur 2 que cela soit `A`). Si par exemple le personnage `A` est choisi, il attaque `B`, puis `B` attaque `A` en retour. Le programme affiche le résultat du tour, à savoir combien de points de vie ont été perdus par chaque personnage. Ceci se répète jusqu'à ce qu'un des personnages devienne inconscient. Le programme affiche alors le vainqueur du combat.