Séquence 5 : Interactions entre objets

Objectif : se placer en situation d'utilisateur d'objets préexistant. Utiliser leurs services tels que documentés dans la documentation API standard (voir annexes). Mettre en oeuvre une application où plusieurs objets coexistent et interagissent.

On souhaite réaliser le jeu suivant : une succession de 20 multiplications maximum est proposée à l'utilisateur. Celui-ci propose un résultat pour chacune. Il dispose d'un délai d'une minute maxi pour l'ensemble du test, et s'il n'a pas répondu aux 20 questions dans le délai imparti le programme s'arrête.

Exemple de sorties :

```
10 X 2 = 20
Bien ! Votre score : 1/20
5 X 10 = 50
Bien ! Votre score : 2/20
3 X 3 = 9
Bien ! Votre score : 3/20
7 X 6 = 42
Bien ! Votre score : 4/20
7 X 6 = 42

....

Bien ! Votre score : 16/20
7 X 7 = 49
Bien ! Votre score : 17/20
10 X 4 = 40
Bien ! Votre score : 18/20
8 X 1 = 8
Bien ! Votre score : 19/20
1 X 6 = 6
Bien ! Votre score : 20/20
Vous avez la note 20/20
```

```
10 X 7 = 70
Bien ! Votre score : 1/20
6 X 7 = 42
Bien ! Votre score : 2/20
2 X 6 = 12
Bien ! Votre score : 3/20
7 X 6 = 42
Bien ! Votre score : 4/20
8 X 10 = 80

....

Echec !Votre score : 9/20
9 X 3 = 27
Bien ! Votre score : 10/20
3 X 3 = 2
Echec !Votre score : 10/20
2 X 6 = 11
Echec !Votre score : 10/20
8 X 5 = 40
Bien ! Votre score : 11/20
7 X 10 = 70
Délai atteint !
```

Pour cet exercice vous utiliserez les classes suivantes contenues dans le package *java.util et java.lang:*

- ✔ *Random* pour disposer d'un générateur d'entiers aléatoires.
- ✔ *Timer* pour disposer de la classe proposant un mécanisme de *callback*.
- ✔ La classe, fournie ci-dessous et directement construite à partir de la classe *TimerTask,* appelée *ExitTask*.
- ✔ La classe *Scanner* permet un accès commode aux nombres saisis au clavier.
- ✔ La classe *System* pour adosser l'objet *Scanner* à l'entrée standard (par *new Scanner(System.in)*), et pour forcer la terminaison du programme lorsque le délai est atteint.

Exercice 1

- La classe *Random* sera sollicitée sur sa capacité à produire des entiers aléatoires dans un intervalle donné ([1-10] dans l'exemple ci-dessus). Repérer, en vous aidant de l'aide en annexe, les méthodes qui seront utiles pour cela.

Correction

La méthode *int nextInt(int)* de *Random* fournit le service désiré.

- En déduire la séquence minimale de code qui permet d'obtenir un entier aléatoire entre 1 et 10

Correction

```
Random random=new Random();
System.out.println(random.nextInt(10)+1);
```

Exercice 2

La classe *Timer* fournit le service permettant d'armer une temporisation (*1 mn* dans l'exemple ci-dessus). A l'issue de ce délai l'objet *Timer* déclenche une méthode. Celle-ci doit posséder la signature *void run()* et appartenir à une classe issue (par héritage) de la classe *TimerTask*. La classe *ExitTask (écrite* ci-après) hérite de *TimerTask* et fournit cette méthode (qui affiche simplement « *Délai atteint ! »*) et qui clôt le programme.

```
class ExitTask extends TimerTask {
   @Override
   public void run() {
      System.out.println("Délai atteint !");
      System.exit(0);
   }
}
```

- Pour que l'objet *Timer* connaisse le délai de temporisation choisi il faut lui passer cette information lors de la création de cet objet ou après. Repérer avec l'aide en ligne les méthodes permettant de passer le délai de temporisation choisi à l'objet *Timer*.

Correction

Plusieurs versions surchargées *de schedule* se font concurrence sur ce point. La plus simple convient et permet le passage des deux arguments à la fois
void schedule(TimerTask task, long delay)

- Pour que l'objet *Timer* connaisse la méthode à appeler à l'expiration du délai il faut aussi lui passer l'objet présentant cette méthode. Le type de cet objet, *ExitTask*, est un sous-type de *TimerTask.* En déduire la séquence de code minimale (réutilisant la classe *ExitTask*) qui permet d'armer à une minute un objet de type *Timer* sur un objet de type *ExitTask.*

Correction

```
ExitTask task=new ExitTask();
Timer timer=new Timer();
timer.schedule(task,60000);
```

- Si l'utilisateur a répondu aux 20 questions dans le délai imparti il convient de désarmer le *timer*. Repérer la méthode de *Timer* permettant de le faire.

Correction

void cancel()

Exercice 3

La saisie au clavier (c'est à dire sur l'entrée standard) peut être faite de plusieurs façons. La classe *Scanner* notamment permet d'opérer commodément. Cette classe peut lire des flux de provenances variées, aussi il faut ici lui indiquer qu'il s'agit d'un flux en provenance de l'entrée standard (La variable statique *System.in* exprime cette entrée standard en Java).

- Repérer dans *Scanner* le service qui permet d'associer un objet *Scanner à* un flux donné.

Correction

Plusieurs variantes surchargées du constructeur proposent ce service, en particulier :
Scanner ***(InputStream source)***

- Repérer dans cette classe la méthode qui permet d'extraire un entier du flux entrant.

Correction

```
int next Int()
```

- En déduire la portion de code minimale permettant d'installer un objet *Scanner* sur l'entrée standard et d'extraire le premier entier de ce flux.

Correction

```
Scanner scanner=new Scanner(System.in);
int num=scanner.nextInt();
```

Exercice 4

- Écrire le programme complet (sous le *main* à l'exception de la classe *ExitTask* qui pourra être placée avant). L'aide en ligne complète concernant *Timer, TimerTask, Random, Scanner* et *System* est disponible sous *http://java.sun.com/javase/6/docs/api/*.

Correction

```
class ExitTask extends TimerTask {
   @Override
   public void run() {
      System.out.println("Délai atteint !");
      System.exit(0);
   }
}

public class Launcher {
   final static int DELAY=60000,MAX_NUM=10;
   public static void main(String[] args) {
      Random random=new Random();
      Timer timer=new Timer();
      ExitTask task=new ExitTask();;
      timer.schedule(task, DELAY);
      int op1,op2,inputValue,score=0,i=0;
      while (i<20) {
         op1=1+random.nextInt(MAX_NUM);
         op2=1+random.nextInt(MAX_NUM);
         System.out.printf("%d X %d = ",op1,op2);
         Scanner scan=new Scanner(System.in);
         inputValue=scan.nextInt();
```

```
        if (inputValue==op1*op2) {
           score++;
           System.out.print("Bien ! ");
        }
        else {
           System.out.print("Echec !");
        }
        System.out.printf("Votre score : %d/20\n",score);
     }
     System.out.printf("Vous avez la note  %d/20\n",score);
     timer.cancel();
  }
}
```

Exercice 5

On souhaite que, lorsque le délai est atteint avant que l'utilisateur ait répondu à la totalité des questions, le score final apparaisse plus clairement que ci-dessus (message réduit à "Délai atteint"). Par exemple on souhaite l'affichage suivant : "Délai atteint ! Votre score final est 11".

- Quel problème pose cette modification ? Comment le résoudre?

Correction

Pb : l'objet *ExitTask* affiche via sa méthode *run* le message *"Délai atteint"* mais ne dispose pas de la valeur du score. puisque cette dernière information est détenue par la classe de lancement (par exemple) sous la forme de la variable locale *score*. C'est un problème typique de l'orienté objet : organiser un passage d'information pertinent entre objets.
La variable *score* pourrait être passée lors de la création de l'objet *ExitTask* mais comme il s'agit d'un type de base c'est un passage par valeur et procédant donc d'une recopie. Cette copie ne pourra bien sûr pas répercuter les évolutions de la variable *score* d'origine (pour la même raison localiser la classe *ExitTask* dans la méthode *main*, pour disposer d'un moyen de récupérer cette valeur locale conduirait au même échec résultant de la sémantique de valeur).

Proposition :
Il faut donc ici disposer d'un objet entrepôt pour cette valeur pour en permettre une lecture/écriture. La classe *Compteur* simple vue en cours conviendrait. Dans le code ci-dessous remarquer le rôle du constructeur *ExitTask* permettant d'installer la dépendance vis à vis d'un *Compteur*.

```
class ExitTask extends TimerTask {
   private Compteur score;
   public ExitTask(Compteur score) { score=this.score; }
   @Override
   public void run() {
      System.out.println("Délai atteint ! Votre score final est "+compteur.getValue());
      System.exit(0);
   }
}

public class Launcher {
   final static int DELAY=60000,MAX_NUM=10;
   public static void main(String[] args) {
      Random random=new Random();
      Timer timer=new Timer();
      Compteur score=new Compteur();
       TimerTask task=new ExitTask(score);;
      timer.schedule(task, DELAY);
      int op1,op2,inputValue,i=0;
      while (i<20) {
         op1=1+random.nextInt(MAX_NUM);
         op2=1+random.nextInt(MAX_NUM);
         System.out.printf("%d X %d = ",op1,op2);
         Scanner scan=new Scanner(System.in);
         inputValue=scan.nextInt();
         if (inputValue==op1*op2) {
            score.up();
            System.out.print("Bien ! ");
         }
         else {
            System.out.print("Echec !");
         }
         System.out.printf("Votre score : %d/20\n",score.getValue());
      }
      System.out.printf("Vous avez la note  finale %d/20\n",score.getValue());
```

```
            timer.cancel();
        }
    }
```

Sequence 6 : Relations entre objets

Objectifs : Relations entre objets. Architecture. Expression en Java. Principe de couplage minimum. Délimitation des responsabilités d'un objet.

Exercice 1

Pour l'application décrite ci-après, imaginer les objets qui vous semblent pertinents, établir leurs sémantiques dans des schémas objet ; puis schématiser les relations de visibilité entre classes.
Il s'agit d'une application qui génère un index de ligne (ou de page peu importe) à partir d'un document donné (3 premières étapes de Booch). Le document est associé à un fichier texte (ou ascii). Ces objets doivent pouvoir être réutilisés dans d'autres applications, il convient donc de les concevoir aussi généraux que possible. Il s'agit d'un exemple plus simple que celui traité en cours, en particulier on ne tient pas compte du tout des questions d'affichage (pas de Vue, Contrôle etc.).N'oubliez pas les principes généraux concernant la modularité.

- ✔ Interface simple.
- ✔ Interface explicite.
- ✔ Créer des objets aussi découplés les uns des autres que possible.

Remarque: Si vous bloquez sur cet exercice vous pouvez vous inspirer de l'exemple de programme de test ci-dessous qui met en oeuvre les objets en question et invoque sur eux les services utiles à l'application. Cette portion de pseudo-code répond indirectement aux questions posées précédemment, aussi essayer d'abord de résoudre le problème sans vous aider de ce code.
Exemple :

```
// Exemple de programme principal en pseudo code
unIndex.créer;
unDocument.créer;
unDocument.attacher( »fichier.txt »);
tant que not unDocument.fin faire
unMot=unDocument.lireMot;
unNumeroLigne=unDocument.numeroLigne;
unIndex.insere(unMot,uneLigne);
unDocument.prochainMot;
fin tant que
// l'index est créé et rempli; il reste à l'exploiter
// pour affichage ou autres...
// La description des services correspondant n'est pas demandée
```

- Proposer un dictionnaire des objets pertinents de cette application

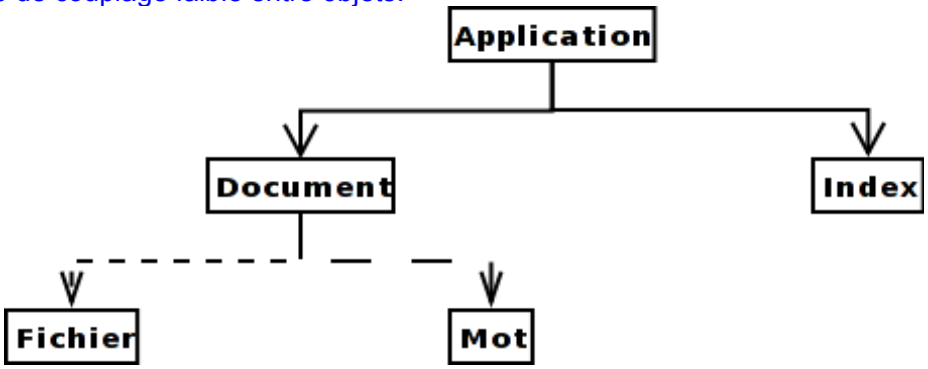
Correction

Index est un objet qui accueille les couples <mot,numéro de page> qu'on lui soumet
Document est un objet qui contient un texte et en offre un principe d'extraction de mots et du numéro de page associé.
Application sera incarnée par la classe de lancement de l'application. Elle sert simplement à mettre en scène et tester les autres objets.
Mot sera préexistante et incarnée par la classe *String*.
L'expression de numéro de page ou de ligne sous forme d'objet semble inutile

- Proposer une architecture pour cette application

Correction

Spontanément les étudiants proposent pour cet exercice un lien de dépendance (souvent bidirectionnel) entre *Document* et *Index*. Expliquer que, comme on a délimité la responsabilité de chacune de ces classes auparavant, ce lien est inutile sur un plan opératoire, mais surtout qu'il vaut mieux qu'il n'y ait aucune relation entre les deux : ainsi chaque classe devient davantage réutilisable dans d'autres contextes puisqu'elle n'est pas dépendante de la présence de l'autre. L'application y gagne et chaque classe y gagne, c'est une illustration du principe de couplage faible entre objets.



- Traduire le programme principal de l'exercice précédent exprimé en pseudo-code en une classe de test Java. Combiner l'ensemble pour obtenir une entité compilable (même s'il elle ne fait rien d'intéressant)

Correction

```
// correction à compléter par la rédaction de Document et Index (uniquement déclaratifs, pas de code)
public class Launcher {
  public static void main(String [] arg) {
    Index index=new Index();
    Document doc=new Document("/home/durand/text1.txt");
    String mot;
    int num;
    while (!doc.isEnd()) {
      mot=doc.getWord();
      num=doc.getNum();
      index.insert(mot,num);
      doc.nextWord();
    }
  }
}
```

Exercice 2

- Imaginer une structure de données internes permettant à index de gérer sa liste de mots

Correction

Cette question anticipe probablement sur le cours mais fait écho au module A31, consacré aux structures de données. Idée : structure de données à clé (par exemple *TreeMap*). La clé est le mot, la valeur est une liste chainée de valeur de numéro. Le code ci-dessous donné à titre d'exemple, mais bien sûr à ce stade on ne demande pas aux étudiants d'être capable d'écrire ça.

```
public class Index {
    private TreeMap<String, ArrayList<Integer>> treeMap=new TreeMap<String, ArrayList<Integer> >();

    public void addWord(String word, int lineNum) {
      List<Integer>  listNum=treeMap.get(word);
      if (listNum!=null) listNum.add(lineNum);
      else {
       ArrayList<Integer> list=new ArrayList<Integer>();
       list.add(lineNum);
       treeMap.put(word,list );
      }
    }

    public String toString() {
      String result="";
      Set<String> set=treeMap.keySet();
      for(String key:set) {
       result+=key;
       for(Integer num:treeMap.get(key)) result+=" "+num;
       result+="\n";
      }
      return result;
    }
 }
```

Exercice 3

- Imaginer, sur le modèle itératif suggéré par la classe Document, les services qui permettent une exploitation de l'index.

Correction

par exemple :
getWord(), getListNum(), isLastWord(), nextWord()
Autre solutions : retourner la structure TreeMap elle même par une méthode getTreeMap par exemple, mais :

- *entorse au prinicpe d'encapsulation des données*
- *impossibilité en Java (au contraire de C++) de délivrer cette structure en lecture seule, donc mise en danger de la cohérence des données internes de l'Index*