TD3: Exceptions
Correction

Exercice 1. Exécution de code

```
class Erreur extends Exception { public int num; }
class Erreur_d extends Erreur { public int code; }
class A {
    public A(int n) throws Erreur_d {
        if (n==1) {
            Erreur_d e = new Erreur_d();
            e.num = 999; e.code = 12;
            throw e ;
        }
    }
}

public class Ex1 {
    public static void main (String args[]){
        try{
            A a = new A(1) ;
            System.out.println ("apres creation a(1)");
        } catch (Erreur e) {
            System.out.println ("** exception Erreur " + e.num);
        }
        System.out.println ("suite main") ;
        try{
            A b = new A(1);
            System.out.println ("apres creation b(1)");
        } catch (Erreur_d e) {
            System.out.println ("** exception Erreur_d "+e.num+" "+e.code);
        } catch (Erreur e) {
            System.out.println ("** exception Erreur " + e.num);
        }
    }
}
```

a. Quels résultats fournit le programme précédent ?

Correction

```
** exception Erreur 999
suite main
** exception Erreur_d 999 12
```

b. Que se passe-t-il

1. si l'on permute l'ordre des deux gestionnaires dans le second bloc try ?

   Correction
   Comme une exception Erreur_d est aussi de type Erreur, la compilation plante en disant : error : exception Erreur_d has already been caught

2. si on remplace le **throws** Erreur_d, Erreur par **throws** Erreur_d dans le constructeur de A ?

Correction
La compilation plante : warning: unreachable **catch** clause

Exercice 2. Classe d'entiers naturels

a. Réaliser une classe permettant de manipuler des entiers naturels (positifs ou nuls) et disposant :
— d'un constructeur à un argument de type **int** ; il générera une exception ErrConst si la valeur de son argument est négative ;
— de méthodes statiques de somme, de différence et de produit de deux naturels ; elles généreront respectivement des exceptions ErrSom, ErrDiff et ErrProd lorsque le résultat ne sera pas représentable ; la limite des valeurs des naturels sera fixée à la plus grande valeur du type **int** ;
— une méthode d'accès getN fournissant sous forme d'un **int** la valeur de l'entier naturel.
On s'arrangera pour que toutes les classes exception dérivent d'une classe ErrNat et pour qu'elles permettent à un éventuel gestionnaire de récupérer les valeurs ayant provoqué l'exception.

b. Écrire deux exemples d'utilisation de la classe :
— l'un se contentant d'intercepter sans discernement les exceptions de type dérivé de ErrNat,
— l'autre qui explicite la nature de l'exception en affichant les informations disponibles.
Les deux exemples pourront figurer dans deux blocs try d'un même programme.

Correction

```
import java.util.ArrayList;

class NotFieldException extends Exception {}

class ErrNat extends Exception {
        public ErrNat() {
                errvals = new ArrayList<Integer>();
        }

        public ErrNat(int n) {
                this();
                errvals.add(n);
        }

        public ErrNat(int n1, int n2) {
                this();
                errvals.add(n1);
                errvals.add(n2);
        }

        public ArrayList<Integer> errvals;
}

class ErrConst extends ErrNat  {
        public ErrConst(int n)
        {
                super(n);
        }
}

class ErrSom extends ErrNat {
        public ErrSom(int n1, int n2) {
                super(n1, n2);
        }
}

class ErrDiff extends ErrNat {
        public ErrDiff(int n1, int n2) {
                super(n1, n2);
```

```
        }
}

class ErrProd extends ErrNat {
        public ErrProd(int n1, int n2) {
                super(n1, n2);
        }
}

class Nat {

        public Nat(int n) throws ErrConst {
                if (n < 0) {
                        throw new ErrConst(n);
                }
                this.n = n;
        }

        public static Nat Somme(Nat a, Nat b) throws ErrSom {
                if (((a.n - maxval) + b.n) > 0)        {
                        throw new ErrSom(a.n, b.n);
                }
                try {
                        Nat c = new Nat(a.n+b.n);
                        return c;
                } catch (ErrNat e) {
                        throw new ErrSom(a.n, b.n);
                }
        }

        public static Nat Prod(Nat a, Nat b) throws ErrProd {
                int an = a.n;
                int bn = b.n;
                int an15 = an % (1 << 15);
                int bn15 = bn % (1 << 15);
                int an30 = (an / (1 << 15)) % (1 << 15);
                int bn30 = (bn / (1 << 15)) % (1 << 15);

                ErrProd e = new ErrProd(a.n, b.n);
                if (an30 * bn30 >= 2)
                        throw e;

                if (an30 * bn15 >= (1 << 16))
                        throw e;

                if (an15 * bn30 >= (1 << 16))
                        throw e;

                int max = 2147483647;

                max -= (an30 * bn30) * (1 << 30);
                max -= (an30 * bn15) * (1 << 15);
                if (max < 0)
                        throw e;
                max -= (an15 * bn30) * (1 << 15); // not managing the possible underflow
                // so probably not correct)
                if (max < 0)
                        throw e;
                max -= (an15 * bn15);
                if (max < 0)
                        throw e;

                try {
                        return new Nat(an*bn);
```

```
                } catch (ErrConst ex) {
                        throw e;
                }
        }
        //
        public static final int maxval = Integer.MAX_VALUE;
        private int n;
}
```