Chapitre 7

Les types énumérés

Connaissances requises

- Définition d'un type énuméré simple (sans champs ni méthodes)
- Utilisation des valeurs d'un type énuméré
- Comparaisons d'égalité entre valeurs d'un type énuméré : opérateur == ou méthode *equals*
- Ordre des valeurs d'un type énuméré : méthodes *compareTo* et *ordinal*
- Conversion en chaînes des constantes d'un type énuméré, avec la méthode *toString*
- Conversion éventuelle d'une chaîne en une valeur d'un type énuméré ; méthode *valueOf*
- Méthode *values* de la classe *Enum*
- Itération sur les constantes d'un type énuméré
- Introduction de champs et de méthodes dans un type énuméré ; cas particulier des constructeurs (transmission d'arguments)

Note : Les types énumérés ne sont disponibles qu'à partir du JDK 5.0.

83 Définition et utilisation d'un type énuméré simple

1. Définir un type énuméré nommé *Couleurs* dont les valeurs sont définies par les identificateurs suivants : *rouge*, *bleu*, *vert*, *jaune*.
2. Déclarer deux variables *c1* et *c2* du type *Couleurs* et leur affecter une valeur.
3. Échanger le contenu de ces deux variables, en s'assurant au préalable que leurs valeurs ne sont pas égales.
4. Regrouper toutes ces instructions dans une petit programme complet (on pourra ajouter des instructions d'affichage des valeurs des variables avant et après échange).

Solution

1. La définition d'un type énuméré en Java utilise une syntaxe de la forme :

   enum NomType { valeur1, valeur2, ... valeurN }

   soit, ici :

   ```
   enum Couleurs { rouge, bleu, vert, jaune }
   ```

   Notez que, bien que l'on emploie le mot-clé *enum* et non *class*, *Couleurs* est à considérer comme un classe particulière. Les valeurs du type (*rouge*, *bleu*, *vert* et *jaune*) en sont des instances finales (non modifiables).

2. La déclaration de variables du type *Couleurs* est classique :

   ```
   Couleurs c1, c2 ;
   ```

   On ne peut affecter à ces variables que des valeurs du type *Couleurs*. Ici, il peut s'agir de l'une des 4 constantes du type : on les nomme en les préfixant du nom de type (*ici Couleurs*) comme dans :

   ```
   c1 = Couleurs.bleu ;   // attention : c1 = bleu serait erroné
   c2 = Couleurs.jaune ;
   ```

3. La comparaison de deux variables de type énuméré peut se faire indifféremment avec l'un des opérateurs == ou *equals*. Rappelons que le premier compare les références des objets correspondants, tandis que le second porte sur les valeurs de ces objets. Mais, comme il n'existe qu'un exemplaire de chaque objet représentant un constante d'un type énuméré, il revient bien au même de comparer leur référence ou leur valeur. De même, on peut utiliser indifféremment *!=* ou *!equals*.

   ```
   if (c1 != c2)            //  ou if (! c1.equals(c2))
   { Couleurs c ;
     c = c1 ;
     c1 = c2 ;
     c2 = c ;
   }
   ```

4. Voici un exemple complet reprenant ces différentes instructions, accompagné d'un exemple d'exécution. On notera qu'il est très facile d'afficher une valeur de type énuméré puisque l'appel implicite à la méthode *toString* pour une instance de type énuméré fournit simplement le libellé correspondant :

```
public class EnumSimple
{ public static void main (String args[])
  { Couleurs c1, c2 ;
    c1 = Couleurs.bleu ;  // attention : c1 = bleu serait erroné
    c2 = Couleurs.jaune ;
    System.out.println ("couleurs avant echange = " + c1 + " " + c2) ;
    if (c1 != c2)     // ou if (! c1.equals(c2))
    { Couleurs c ;
      c = c1 ;
      c1 = c2 ;
      c2 = c ;
    }
    System.out.println ("couleurs apres echange = " + c1 + " " + c2) ;
  }
}
enum Couleurs {rouge, bleu, vert, jaune }
```

```
couleurs avant echange = bleu jaune
couleurs apres echange = jaune bleu
```

84 Itération sur les valeurs d'un type énuméré

On suppose qu'on dispose d'un type énuméré nommé *Suite*. Écrire un programme qui en affiche les différents libellés. Par exemple, si *Suite* a été défini ainsi (notez l'emploi du libellé *ut*, car *do* n'est pas utilisable puisqu'il s'agit d'un mot-clé) :

```
enum Suite { ut, re, mi, fa, sol, la, si }
```

Le programme affichera :

```
Liste des valeurs du type Suite :
ut
re
mi
fa
sol
la
si
```

Solution

On peut facilement itérer sur les différentes valeurs d'un type énuméré à l'aide de la boucle dite *for... each*, introduite par le JDK 5.0. Il faut cependant au préalable créer un tableau des valeurs du type en utilisant la méthode *values* de la classe *Enum* ; l'expression *Suite.values()* représente un tableau formé des différentes valeurs du type *Suite*. En définitive, voici le programme voulu ; il fonctionne quelle que soit la définition du type *Suite* :

```
public class TstSuite
{ public static void main (String args[])
  { System.out.println( "Liste des valeurs du type Suite : " ) ;
    for (Suite s : Suite.values() )
      System.out.println (s) ;     // appel implicite de toString ()
  }
}
enum Suite { ut, re, mi, fa, sol, la, si }
```

85 Accès par leur rang aux valeurs d'un type énuméré (1)

On suppose qu'on dispose d'un type énuméré nommé *Suite*. Ecrire un programme qui :

- affiche le nombre de valeurs du type,
- affiche les valeurs de rang impair,
- affiche la dernière valeur du type.

Solution

Une démarche simple consiste à créer un tableau des valeurs du type, à l'aide de la méthode *values* de la classe *Enum*. Il suffit ensuite d'exploiter classiquement ce tableau pour obtenir les informations voulues :

```
public class TstValues
{ public static void main (String args[])
  { // On crée un tableau des valeurs du type, à l'aide de la méthode values
   Suite[] valeurs = Suite.values () ;
   int nbVal = valeurs.length ;
   System.out.println ("le type Suite comporte " + nbVal + " valeurs" ) ;
   System.out.println ("valeurs de rang impair = ") ;
   for (int i =0 ; i < nbVal ; i+=2)
      System.out.println (valeurs[i]) ;
   System.out.println ("derniere valeur du type : ") ;
   System.out.println (valeurs[nbVal-1]) ;
 }
enum Suite { ut, re, mi, fa, sol, la, si }
```

```
le type Suite comporte 7 valeurs
valeurs de rang impair =
ut
mi
sol
si
derniere valeur du type :
si
```

On notera que le programme n'est pas protégé contre le risque que le type *Suite* ne comporte aucun élément.

86 Lecture de valeurs d'un type énuméré

On suppose qu'on dispose d'un type énuméré nommé *Suite*. Écrire un programme qui lit une chaîne au clavier et qui indique si cette chaîne correspond ou non à un libellé du type et qui, le cas échéant, en affiche le rang dans les valeurs du type.

Solution

A priori, toute classe d'énumération dispose d'une méthode *valueOf* qui effectue la conversion inverse de *toString*, à savoir : convertir une chaîne en une valeur du type énuméré correspondant. Cependant, si la chaîne en question ne correspond à aucune valeur du type, on aboutit à une exception qui doit alors être interceptée, sous peine de voir le programme s'interrompre. Ici, nous vous proposons une démarche, moins directe, mais ne comportant plus de risque d'exception, à savoir : parcourir chacune des valeurs du type énuméré (à l'aide du tableau fourni par la méthode *values*) en comparant sa conversion en chaîne (*toString*) avec la chaîne fournie au clavier.

```
public class LectureEnum
{ public static void main (String args[])
  { String chSuite ;
    System.out.print("Donnez un libelle de l'enumeration Suite : ");
    chSuite = Clavier.lireString () ;
    boolean trouve = false ;
    for (Suite j : Suite.values())
      { if (chSuite.equals(j.toString() ) )
        { trouve = true ;
          int numSuite = j.ordinal() ;
          System.out.println(chSuite  + " correspond a la valeur de rang  "
                                      + (numSuite+1) + " du type Suite" );
        }
      }
```

```
    if (!trouve) System.out.println (chSuite
                        + " n'appartient pas au type Suite") ;
  }
}
enum Suite {ut, re, mi, fa, sol, la, si }
```

```
Donnez un libelle de l'enumeration Suite : Re
Re n'appartient pas au type Suite
```

```
Donnez un libelle de l'enumeration Suite : mi
mi correspond a la valeur de rang  3 du type Suite
```

87 Ajout de méthodes et de champs à une énumération (1)

Définir un type énuméré nommé *Mois* permettant de représenter les douze mois de l'année, en utilisant les noms usuels (*janvier*, *fevrier*, *mars*...) et en associant à chacun le nombre de jours correspondants. On ne tiendra pas compte des années bisextiles.

Écrire un petit programme affichant ces différents noms avec le nombre de jours correspondants comme dans :

```
janvier comporte 31 jours
fevrier comporte 28 jours
mars comporte 31 jours
   .....
octobre comporte 31 jours
novembre comporte 30 jours
decembre comporte 31 jours
```

Solution

Java vous permet de doter un type énumération de champs et de méthodes, comme s'il s'agissait d'une classe. Certaines de ces méthodes peuvent être des constructeurs ; dans ce cas, il est nécessaire d'utiliser une syntaxe spéciale dans la définition du type énuméré pour fournir les arguments destinés au constructeur, en association avec le libellé correspondant.

Voici comment nous pourrions définir notre type *Mois*, en le munissant :

- d'un champ *nj* destiné à contenir le nombre de jours d'un mois donné,
- d'un constructeur recevant en argument le nombre de jours du mois,
- d'une méthode *nbJours* fournissant le nombre de jours associé à une valeur donnée.

```
enum Mois
{ janvier (31),   fevrier (28), mars (31),     avril (30),
  mai (31),       juin (30),    juillet (31),  aout (31),
  septembre (30), octobre (31), novembre (30), decembre (31) ;
  private Mois (int n) // constructeur (en argument, nombre de jours du mois)
  { nj = n ; ;
  }
  public int nbJours () { return nj ; }
  private int nj ;
}
```

Notez les particularités de la syntaxe :

- présence d'arguments pour le constructeur,
- présence d'un point-virgule séparant l'énumération des valeurs du type des déclarations des champs et méthodes.

Voici un petit programme fournissant la liste voulue.

```
public class TstMois
{ public static void main (String args[])
  { for (Mois m : Mois.values() )
       System.out.println ( m + " comporte " + m.nbJours() + " jours") ;
  }
}
```

88 Ajout de méthodes et de champs à une énumération (2)

Compléter la classe *Mois* précédente, de manière à associer à chaque nom de mois :

- un nombre de jours,
- une abréviation de trois caractères (*jan*, *fev*...),
- le nom anglais correspondant.

Écrire un petit programme affichant ces différentes informations sous la forme suivante :

```
jan = janvier = january - 31 jours
fev = fevrier = february - 28 jours
mar = mars = march - 31 jours
   .....
oct = octobre = october - 31 jours
nov = novembre = november - 30 jours
dec = decembre = december - 31 jours
```

Solution

Il suffit d'adapter l'énumération *Mois* de l'exercice précédent de la façon suivante :

- introduction de nouveaux champs *abrege* et *anglais* pour y conserver les informations relatives au nom abrégé et au nom anglais,
- ajout de méthodes *abreviation* et *nomAnglais* fournissant chacune de ces informations,
- adaptation du constructeur pour qu'il dispose cette fois de trois arguments.

```
enum Mois2
{ janvier   (31, "jan", "january"),     fevrier  (28, "fev", "february"),
  mars      (31, "mar", "march"),       avril    (30, "avr", "april"),
  mai       (31, "mai", "may"),         juin     (30, "jun", "june"),
  juillet   (31, "jul", "july"),        aout     (31, "aou", "august"),
  septembre (30, "sep", "september"),   octobre  (31, "oct", "october"),
  novembre  (30, "nov", "november"),    decembre (31, "dec", "december") ;
  private Mois2 (int n, String abrev, String na)
  { nj = n ;
    abrege = abrev ;
    anglais = na ;
  }
  public int nbJours () { return nj ; }
  public String abreviation ()
  { return abrege ;
  }
  public String nomAnglais ()
  { return anglais ;
  }
  private int nj ;
  private String abrege ;
  private String anglais ;
}

public class TstMois2
{ public static void main (String args[])
  { for (Mois2 m : Mois2.values() )
       System.out.println ( m.abreviation() + " = " + m + " = "
                          +m.nomAnglais() + " - " + m.nbJours() + " jours")  ;
  }
}
```

89 Synthèse : gestion de résultats d'examens

On se propose d'établir les résultats d'examen d'un ensemble d'élèves. Chaque élève sera représenté par un objet de type *Eleve*, comportant obligatoirement les champs suivants :

- le nom de l'élève (type *String*),
- son admissibilité à l'examen, sous forme d'une valeur d'un type énuméré comportant les valeurs suivantes : N (non admis), P (passable), AB (*Assez bien)*, B (*Bien*), TB (*Très bien)*.

Idéalement, les noms des élèves pourraient être contenus dans un fichier. Ici, par souci de simplicité, nous les supposerons fournis par un tableau de chaînes placé dans le programme principal.

On demande de définir convenablement la classe *Eleve* et d'écrire un programme principal qui :

- pour chaque élève, lit au clavier 3 notes d'examen, en calcule la moyenne et renseigne convenablement le champ d'admissibilité, suivant les règles usuelles :
  - moyenne < 10 : Non admis
  - 10 <= moyenne <12 : Passable
  - 12 <= moyenne <14 : Assez bien
  - 14 <= moyenne <16 : Bien
  - 16 <= moyenne : Très bien
- affiche l'ensemble des résultats en fournissant en clair la mention obtenue.

  Voici un exemple d'exécution d'un tel programme :

```
donnez les trois notes de l'eleve Dutronc
11.5
14.5
10
donnez les trois notes de l'eleve Dunoyer
9.5
10.5
9
donnez les trois notes de l'eleve Lechene
14.5
12
16.5
donnez les trois notes de l'eleve Dubois
6
14
11
donnez les trois notes de l'eleve Frenet
17.5
14
18.5
```

```
Resultats :
Dutronc - Assez bien
Dunoyer - Non admis
Lechene - Bien
Dubois - Passable
Frenet - Tres bien
```

Solution

L'énoncé nous impose la définition du type énuméré contenant les différents résultats possibles de l'examen. On notera qu'on nous demande d'afficher ces résultats sous une forme « longue », par exemple *Passable* et non simplement *P*. Nous associerons donc un texte à chacune des valeurs de notre type énuméré, en exploitant la possibilité de doter un tel type de méthodes, à savoir ici :

- un constructeur recevant en argument le texte associé à la valeur,
- une méhtode nommée *details*, permettant de trouver ce texte à partir d'une valeur.

Voici ce que pourrait être la définition de ce type énuméré :

```
enum Mention
{ NA ("Non admis"), P ("Passable"), AB ("Assez bien"),
                B ("Bien"), TB ("Tres bien"), NC ("Non connu") ;
  private Mention (String d)
  { mentionDetaillee = d ;
  }
  public String details ()
  { return mentionDetaillee ;
  }
  private String mentionDetaillee ;
}
```

Un champ privé nommé *mentionDetaillee* nous sert à conserver le texte associé à chaque valeur.

Notez que, pour des questions de sécurité, nous avons prévu une valeur supplémentaire (*NC*) correspondant à un résultat non connu, avec laquelle se trouvera automatiquement initialisée (par le constructeur) toute variable du type *Mention*,

Nous avons prévu d'utiliser deux méthodes statiques :

- *double moyenne (String n)* qui demande de fournir trois notes pour le nom *n* et qui en calcule la moyenne,
- *Mention resul (double m)* qui fournit la mention correspondant à une moyenne donnée *m*.

Voici ce que pourrait être le programme demandé :

```
public class Examen
{ public static void main (String args[])
  { String noms[] = { "Dutronc", "Dunoyer", "Lechene", "Dubois", "Frenet" } ;
        // creation du tableau d'eleves
    int nel = noms.length ;
```

```
      Eleve eleves [] = new Eleve [nel] ;
      for (int i=0 ; i<nel ; i++)
         eleves [i] = new Eleve (noms[i]) ;

      // lecture des notes et détermination du résultat de chaque élève
    for (Eleve el : eleves)
      { double moy = moyenne (el.getNom()) ;
        el.setResul ((resul(moy))) ;
      }

      // affichage résultats
    System.out.println ("Resultats : ") ;
    for (Eleve el : eleves)
      System.out.println (el.getNom() + " - " + el.getResul().details())  ;
  }

     // méthode qui demande au clavier trois notes pour un nom donne
     // et qui fournit en retour la moyenne correspondante
  static public double moyenne (String n)
  { System.out.println ("donnez les trois notes de l'eleve " + n) ;
    double som = 0. ;
    for (int i=0 ; i<3 ; i++)
    { double note = Clavier.lireDouble() ;
      som += note ;
    }
    double moyenne = som / 3. ;
    return moyenne ;
  }

    // méthode qui définit la mention en fonction de la moyenne
  static public Mention resul (double m)
  { if ( m<10. ) return Mention.NA ;
    if ( m<12.0) return Mention.P ;
    if ( m<14.0) return Mention.AB ;
    if ( m<16.0) return Mention.B ;
    return Mention.TB ;
  }
}

class Eleve
{ public Eleve (String n)
  { nom = n ;
    resul = Mention.NC ;   // valeur par défaut
  }
  public void setResul (Mention r)
  { resul = r ;
  }
  public Mention getResul()
  { return resul ;
  }
```

```
  public String getNom()
  { return nom ;
  }
  private String nom ;
  private Mention resul ;
}

enum Mention
{ NA ("Non admis"), P ("Passable"), AB ("Assez bien"),
                B ("Bien"), TB ("Tres bien"), NC ("Non connu") ;
  private Mention (String d)
  { mentionDetaillee = d ;
  }
  public String details ()
  { return mentionDetaillee ;
  }
  private String mentionDetaillee ;
}
```

Chapitre 8

Les exceptions

Connaissances requises

- Déclenchement d'une exception avec *throw*
- Bloc *try* , écriture d'un gestionnaire d'exception
- Transmission d'informations au gestionnaire d'exception
- Règles de choix du gestionnaire d'exception
- Cheminement d'une exception
- Clause *throws*
- Bloc *finally*
- Redéclenchement d'une exception
- Exceptions standard

90 Déclenchement et traitement d'une exception

Réaliser une classe *EntNat* permettant de manipuler des entiers naturels (positifs ou nuls). Pour l'instant, cette classe disposera simplement :

- d'un constructeur à un argument de type *int* qui générera une exception de type *ErrConst* (type classe à définir) lorsque la valeur reçue ne conviendra pas,
- d'une méthode *getN* fournissant sous forme d'un *int*, la valeur encapsulée dans un objet de type *EntNat*.

Écrire un petit programme d'utilisation qui traite l'exception *ErrConst* en affichant un message et en interrompant l'exécution.

Solution

Le constructeur de la classe *EntNat* doit donc déclencher une exception de type *ErrConst* lorsque la valeur reçue par son constructeur est négative. Ici, la classe *ErrConst* peut être réduite à sa plus simple expression, à savoir ne comporter ni champs ni méthodes. La définition de *EntNat* pourrait se présenter ainsi :

```
class EntNat
{ public EntNat (int n) throws ErrConst
  { if (n<0) throw new ErrConst() ;
    this.n = n ;
  }
  public int getN () { return n ; }
  private int n ;
}
class ErrConst extends Exception
{}
```

On notera qu'en l'absence de la clause *throws ErrConst* dans l'en-tête du constructeur de *EntNat*, on obtiendrait une erreur de compilation. D'autre part, il est indispensable que la classe *ErrConst* dérive de la classe *Exception* (le compilateur s'assure bien que l'objet mentionné à *throw* est d'un type compatible avec *Exception*).

Voici un programme d'utilisation dans lequel nous traitons l'exception *ErrConst* en incluant les instructions concernées dans un bloc *try* que nous faisons suivre d'un gestionnaire introduit par *catch(ErrConst e)*. Comme demandé, nous y affichons un message (**** erreur construction ****) et nous mettons fin à l'exécution par l'appel de *System.exit*.

```
public class TstEntNat
{ public static void main (String args[])
  { try
    { EntNat n1 = new EntNat(20) ;
      System.out.println ("n1 = " + n1.getN()) ;
      EntNat n2 = new EntNat(-12) ;
      System.out.println ("n2 = " + n2.getN()) ;
    }
    catch (ErrConst e)
    { System.out.println ("*** erreur construction ***") ;
      System.exit (-1) ;
    }
  }
}
```

```
n1 = 20
*** erreur construction ***
```

91 Transmission d'information au gestionnaire

Adapter la classe *EntNat* de l'exercice et le programme d'utilisation de manière à disposer dans le gestionnaire d'exception du type *ErrConst* de la valeur fournie à tort au constructeur.

Solution

Cette fois, nous prévoyons, dans la classe *ErrConst*, un champ *valeur* destiné à conserver la valeur avec laquelle on a tenté de construire à tort un entier naturel. La façon la plus simple d'attribuer une valeur à ce champ consiste à le faire lors de la création de l'objet de type *ErrConst*, en la transmettant au constructeur. Ici, nous avons fait de *valeur* un champ privé, de sorte que nous dotons notre classe *ErrConst* d'une méthode d'accès *getValeur*. Voici la nouvelle définition de nos classes *EntNat* et *ErrConst* :

```
class EntNat
{ public EntNat (int n) throws ErrConst
  { if (n<0)  throw new ErrConst(n) ;
    this.n = n ;
  }
  public int getN () { return n ; }
  private int n ;
}
class ErrConst extends Exception
{ public ErrConst (int valeur) { this.valeur = valeur ; }
  public int getValeur() { return valeur ; }
  private int valeur ;
}
```

Dans notre programme d'utilisation, nous devons récupérer la valeur coupable dans le gestionnaire d'exception. Il nous suffit pour cela de recourir à la méthode *getValeur* :

```
public class TstEntN1
{ public static void main (String args[])
  { try
    { EntNat n1 = new EntNat(20) ;
      System.out.println ("n1 = " + n1.getN()) ;
      EntNat n2 = new EntNat(-12) ;
      System.out.println ("n2 = " + n2.getN()) ;
    }
    catch (ErrConst e)
    { System.out.println ("*** tentative construction naturel avec "
                          + e.getValeur() + " ***") ;
      System.exit (-1) ;
    }
  }
}
```

```
n1 = 20
*** tentative construction naturel avec -12 ***
```

Remarque

En pratique, on se permettra souvent de ne pas appliquer le principe d'encapsulation à des champs tels que *valeur*. Ainsi, en le déclarant *public*, on pourra se passer de la méthode *getValeur* et écrire directement dans le gestionnaire :

```
System.out.println ("*** tentative construction naturel avec "
                    + e.valeur + " ***") ;
```

92 Cheminement des exceptions

Que produit le programme suivant lorsqu'on lui fournit en donnée[a] :

- la valeur 0,
- la valeur 1,
- la valeur 2.

```
class Except extends Exception
{ public Except (int n) { this.n = n ; }
  public int n ;
}
```

a. Pour lire un entier au clavier, il utilise la méthode *lireInt* de la classe *Clavier* fournie sur le site Web.

```
public class Chemin
{ public static void main (String args[])
  { int n ;
    System.out.print ("donnez un entier : ") ; n = Clavier.lireInt() ;
    try
    { System.out.println ("debut premier bloc try") ;
      if (n!=0) throw new Except (n) ;
      System.out.println ("fin premier bloc try") ;
    }
    catch (Except e)
    { System.out.println ("catch 1 - n = " + e.n) ;
    }
    System.out.println ("suite du programme") ;
    try
    { System.out.println ("debut second bloc try") ;
      if (n!=1) throw new Except (n) ;
      System.out.println ("fin second bloc try") ;
    }
    catch (Except e)
    { System.out.println ("catch 2 - n = " + e.n) ; System.exit(-1) ;
    }
    System.out.println ("fin programme") ;
  }
}
```

Solution

Ici, il faut simplement savoir que lorsque le gestionnaire d'exception ne comporte pas d'arrêt de l'exécution (ou d'instruction *return*), l'exécution se poursuit à l'instruction suivant le dernier gestionnaire associé au bloc *try*.

En définitive, voici quels seront les trois exemples d'exécution correspondant aux trois réponses proposées :

```
donnez un entier : 0
debut premier bloc try
fin premier bloc try
suite du programme
debut second bloc try
catch 2 - n = 0
```

```
donnez un entier : 1
debut premier bloc try
catch 1 - n = 1
suite du programme
debut second bloc try
fin second bloc try
fin programme
```

```
donnez un entier : 2
debut premier bloc try
```

```
catch 1 - n = 2
suite du programme
debut second bloc try
catch 2 - n = 2
```

93 Cheminement des exceptions et choix du gestionnaire

Quels résultats fournit ce programme ?

```
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 Chemin1
{
  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) ;
        }
      }
    }
```

Que se passe-t-il si l'on inverse l'ordre des deux gestionnaires dans le second bloc *try* ?

Solution

Dans le premier bloc *try*, l'appel du constructeur de *A* déclenche une exception de type *Erreur_d*. Celle-ci est traitée par l'unique gestionnaire relatif au type *Erreur*, lequel convient effectivement puisque *Erreur_d* dérive de *Erreur*. Dans le second bloc *try*, on déclenche la même exception mais, cette fois, deux gestionnaires lui sont associés. Le premier trouvé convient et c'est donc lui qui est exécuté. En définitive, on obtient les résultats suivants :

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

Notez bien qu'ici, les messages *apres creation...* ne sont pas affichés puisque les deux blocs *try* sont interrompus auparavant.

Si l'on inverse l'ordre des deux gestionnaires dans le second bloc *try*, on obtient une erreur de compilation car le second n'a aucune chance d'être sélectionné.

94 Cheminement des exceptions

Que fait ce programme[a] ?

```
class Positif
{ public Positif (int n) throws ErrConst
  { if (n<=0) throw new ErrConst() ;
  }
}
class ErrConst extends Exception
{}
public class TstPos
{ public static void main (String args[])
  { System.out.println ("debut main") ;
    boolean ok = false ;
```

a. Pour lire un entier au clavier, il utilise la méthode *lireInt* de la classe *Clavier* fournie sur le site Web.

```
        while (!ok)
        { try
          { System.out.print ("donnez un entier positif : ") ;
            int n = Clavier.lireInt() ;
            Positif ep = new Positif (n) ;
            ok = true ;
          }
          catch (ErrConst e)
          { System.out.println ("*** erreur construction ***") ;
          }
        }
        System.out.println ("fin main") ;
      }
    }
```

Solution

Dans la boucle *while* de la méthode *main*, on lit un nombre entier qu'on transmet au constructeur de *Positif*. Si la valeur qu'il reçoit n'est pas strictement positive, il déclenche une exception de type *ErrConst*. Celle-ci est traitée dans le gestionnaire associé au bloc *try*, lequel se contente d'afficher un message (**** erreur construction ****). Après l'exécution de ce gestionnaire, on passe à l'instruction suivant le bloc *try*, c'est-à-dire ici au test de poursuite de la boucle *while*, basée sur la valeur de *ok*. On constate que la boucle se poursuit jusqu'à ce que la valeur de *n* soit effectivement positive. Dans ce cas, en effet, la construction de *ep* se déroule normalement et l'on exécute l'instruction *ok=true*.

Voici un exemple d'exécution de ce programme, dans lequel on déclenche à deux reprises l'exception *ErrConst* :

```
debut main
donnez un entier positif : -5
*** erreur construction ***
donnez un entier positif : 0
*** erreur construction ***
donnez un entier positif : 4
fin main
```

Voici un autre exemple dans lequel aucune exception n'a été déclenchée :

```
debut main
donnez un entier positif : 5
fin main
```

Remarque

En général, il n'est pas conseillé d'employer le mécanisme de gestion des exceptions pour contrôler le déroulement d'une boucle comme nous le faisons ici. Cependant, cette démarche pourra s'avérer utile dans quelques rares circonstances, notamment pour lire un fichier séquentiel ; dans ce cas, on se basera sur l'exception *EOFException*.

95 Instruction return dans un gestionnaire

Que fournit le programme suivant ?

```
class Erreur extends Exception
{}
class A
{ public A(int n) throws Erreur
  { if (n==1) throw new Erreur() ;
  }
}
public class Chemin2
{ public static void main (String args[])
  { f(true) ;  System.out.println ("apres f(true)") ;
    f(false) ; System.out.println ("apres f(false)") ;
  }
  public static void f(boolean ret)
  { try
    { A a = new A(1) ;
    }
    catch (Erreur e)
    { System.out.println ("** Dans f - exception Erreur " ) ;
      if (ret) return ;
    }
    System.out.println ("suite f") ;
  }
}
```

Solution

Les deux appels de *f* déclenchent une exception à la construction de *a*. Toutefois, dans le premier cas, le gestionnaire est amené à exécuter une instruction *return*, ce qui met fin à l'exécution de *f*, sans que l'instruction suivant le bloc *try* (affichage de *suite f*) ne soit exécutée. En revanche, elle l'est bien dans le second cas où le gestionnaire se termine naturellement, sans qu'aucune instruction *return* ou *exit* n'ait été exécutée.

En définitive, le programme fournit les résultats suivants :

```
** Dans f - exception Erreur
apres f(true)
** Dans f - exception Erreur
suite f
apres f(false)
```

96 Redéclenchement d'une exception et choix du gestionnaire

Que fournit ce programme ?

```
class Erreur  extends Exception {}
class Erreur1 extends Erreur    {}
class Erreur2 extends Erreur    {}
class A
{ public A(int n) throws Erreur
  { try
    { if (n==1) throw new Erreur1() ;
      if (n==2) throw new Erreur2() ;
      if (n==3) throw new Erreur() ;
    }
    catch (Erreur1 e)
    { System.out.println ("** Exception Erreur1 dans constructeur A") ;
    }
    catch (Erreur e)
    { System.out.println ("** Exception Erreur dans constructeur A") ;
      throw (e) ;
    }
  }
}
public class Redecl
{ public static void main (String args[])
  { int n ;
    for (n=1 ; n<=3 ; n++)
    { try
      { A a = new A(n) ;
      }
      catch (Erreur1 e)
      { System.out.println ("*** Exception Erreur1 dans main") ;
      }
      catch (Erreur2 e)
      { System.out.println ("*** Exception Erreur2 dans main") ;
      }
      catch (Erreur e)
      { System.out.println ("*** Exception Erreur dans main") ;
      }
      System.out.println ("--------------") ;
    }
    System.out.println ("fin main") ;
  }
}
```