Université Ferhat Abbas Sétif 1

Faculté des Sciences- Département Informatique

Polycopie de cours :

Programmation Orientée-Objet

Dr. Lyazid TOUMI
Maître de conférences classe -B- en Informatique
lyazid.toumi@univ-setif.dz

1er Juillet 2018

Table des matières

Livres de références

1. Java the complete reference, 5 Edition , Claude Delannoy, édition Eyrolles, 2008
2. Programmer en Java, 9 edition, Herbert Schildt, Oracle Press, 2014

Langage Java

Histoire

"Écrit une fois, exécuté partout !"

Le langage Java a été développé par *Sun Microsystems* en 1991 dans le cadre d'un projet de recherche visant à développer des logiciels pour des appareils électroniques grand public, tels que téléviseurs, magnétoscopes, grille-pain et autres catégorie de machines que vous pouvez l'acheter dans n'importe quel grand magasin. Les objectifs de Java à cette époque devaient être petits, rapides, efficaces et facilement portables sur une large gamme de périphériques. Ce sont ces mêmes objectifs qui ont fait de Java un langage idéal pour distribuer des programmes exécutables via le Web, et aussi un langage de programmation polyvalent pour développer des programmes facilement utilisables et portables sur différentes plateformes.

Le langage Java a été utilisé dans plusieurs projets au sein de *Sun*, mais n'a pas reçu beaucoup d'attention commerciale jusqu'à ce qu'il soit associé à *HotJava*. *HotJava* a été écrit en 1994 en temps record, à la fois comme un moyen de téléchargement et d'exécution des applets. Cependant, parce que le langage Java existe depuis plusieurs années et a été utilisé pour plusieurs projets, le langage lui-même est assez stable et robuste et ne changera probablement pas excessivement.

Le langage Java a la particularité d'être compilé en byte-codes au lieu d'objets tels que le C/C++. Le byte-codes sont des instructions assembleur d'un processeur virtuel, et donc indépendant du processeur physique de la machine. Un programme Java est compilé de façon abstraite, peut alors s'exécuter sur n'importe quelle plateforme contient une machine virtuelle Java. Cette caractéristique lui permet de survivre dans dans n'importe quelle machine, serveur ou stations connectés via Internet.

Conçus par une équipe dirigée par *James Gosling*, le langage *Java* est une synthèse de plusieurs langages. Le choix de la syntaxe s'est porté sur celle de *C/C++*, pour faciliter le passage des programmeurs vers Java. La structure de Java s'inspire de langages purement objet comme l'ancêtre *Smalltalk*. Il y a eu aussi des emprunts au langage *Ada*. Le choix initial a été guidé par des soucis de simplicité, de rigueur et par conséquents ont conduit à écarter toutes les fonctionnalités du *C++* qui font la difficulté de ce dernier. Cette simplification conceptuelle, ainsi que la rigueur du compilateur, permet de réduire le temps de débogage.

Principales caractéristiques, en bref ..

— Java est un langage complet, qui permet également d'écrire des programmes classiques (et pas seulement des applets dans des pages HTML), comme n'importe quel langage informatique, du type Pascal ou C++.

— Son atout primordial est son universalité (indépendance des plateformes matérielles). Cette indépendance est située à 2 niveaux :

1. Source est portable : vous n'aurez pas besoin de le recompiler quand vous changez le système.
2. Fichier binaire résultat de la compilation est lui aussi indépendant. En effet cette compilation ne génère pas de code machine mais un pseudo-code universel, le Java byte-code, qui ressemble à du langage machine d'un processeur virtuel, en ce sens que ce code n'est intégré dans aucun processeur existant.

   Ce code compilé est inclus dans les fichiers *.class*. Ceux-ci peuvent être distribués sur le réseau Internet, sous forme de petits programmes, les applets.
3. l'exécution seule nécessite un interpréteur Java, appelé JVM (Java Virtual Machine).
   Seule cette JVM dépend étroitement de la plateforme, pour accélérer l'exécution la JVM est intégrée directement sur le système d'exploitation.

— Java est un langage orienté objet utilisé pour développer des grandes applications tels que ORACLE, ANDROID et LINKEDIN.

Chapitre 1

Classes et objets

Au début, nous présentons les notions de bases de la programmation objet. Nous commençons par présenter la syntaxe pour déclarer une classe, ensuite nous vous présentons les instructions nécessaire, qui vous permet de créer des objets à partir d'une classe quelconque, ce qui nous guidera à introduire le principe de référencement à un objet.

Nous vous introduisons par la ensuite la notion de constructeur, ainsi que son importance dans le concept **Objet**. Nous verrons par la suite les opérations sur les objets et en quoi elles se diffèrent des variables de type primitif. Les propriétés d'une méthode (arguments, variables locales,...) seront ensuite étalés.

1.1 Notion de classe

Afin de vous habituer avec les notions de bases concernant la définition d'une classe, d'un objet ou bien comment encapsuler les données. Nous proposons la classe *Personne* comme un exemple illustratif. Cette classe contient des attributs et des méthodes que nous les détaillerons dans cette section.

1.1.1 Définition de la classe Personne

La classe *Personne* est destinée à représenter les informations d'une personne. La syntaxe utilisée par Java pour définir une classe est la suivante :

```
public class Personne
{
    // instructions pour définir les attributs
    //         et les méthodes de la classe Personne
}
```

Plus loin dans ce chapitre nous présentons le rôle exact du mot-clé *public*. Pour l'instant, sachez simplement que ce mot-clé est utilisé limiter l'accès à la classe. Dans notre exemple le mot-clé *public* permet à d'autres classes d'accéder à la classe *Personne*. Dans l'absence du mot clé *public*, l'accès à la classe *Personne* serait limité seulement aux classes voisines, c-à-d, les classes du même répertoire du stockage sur le disque, plus tard sera appelé *package*.

Définition des attributs

Une *Personne* est défini par son nom et prénom, ainsi par son age. Les deux premiers attributs de type *String* et le dernier de type *int*.

La déclaration des attributs se fait comme suite :

```
private String nom;
private String prenom;
private int age;
```

Nous remarquons la présence du mot-clé *private* qui précise que les attributs nom, prenom et age ne sont accessibles qu'à partir de l'intérieur de la classe *Personne* via des méthodes de cette classe, ce paradigme est appelé encapsulation des données.

Définition des méthodes

Pour compléter notre exemple, nous proposons deux méthodes à définir :

— *initialiser* pour affecter des valeurs aux attributs nom, prenom et age,

— *afficher* pour afficher en sortie les valeurs des attributs.

Une méthode est similaire à une procédure ou une fonction dans un langage procédural, comme le langage *C*. Une méthode est constituée d'une en-tête et d'un bloc. La syntaxe de la méthode initialiser est présentée comme suite :

```
public void initialiser (String n, String p, int a)
{
  nom=n;
  prenom=p;
  age=a;
}
```

L'entête d'une méthode se précise comme suite :

— **le nom de la méthode :** dans notre cas *initialiser* ;

— **le mode de restriction :** c'est obligatoire d'utiliser un accès public pour que cette méthode soit accessible ; nous avons déjà vu précédemment le mot-clé *private* dans la présentation des attributs ; nous aurons l'occasion de revenir en détails sur ces problèmes de restriction plus loin.

— **les arguments :** sont fournis à la méthode lors de son appel. Nous avons choisi de donner les noms : n, p et a, aux arguments de passage de paramètres.

— **le type de la valeur de retour :** une méthode est similaire à une fonction, elle peut fournir ou non un résultat en retour, c'est-à-dire se comporter comme une fonction ou comme une procédure dans un langage procédural. Dans notre cas les méthodes proposées ne fournissent aucun résultat, pour cela nous avons préciser le mot-clé *void* dans la valeur de retour.

Si nous observons le corps de la méthode *initialiser*, nous rencontrons notre première affectation :

```
nom=n;
```

Le symbole *n* présente la valeur reçue par passage de paramètre. Quant au *nom*, il ne s'agit ni d'un argument de passage, ni d'une variable locale de la méthode. En fait, le *nom* désigne un attribut d'un objet de type *Personne* qui sera destiné par l'appel de la méthode *initialiser*.

Maintenant nous avons la définition intégrale de la classe *Personne* :

```
public class Personne
{
  private String nom ;
  private String prenom ;
  private float age;
  public void initialiser (String n, String p, int a)
   {
     nom=n ;
     prenom=p ;
     age=a;
   }
  public void afficher ()
   {
     System.out.print ("Je␣suis␣"+nom +"␣"+ prenom);
     System.out.println("␣age:␣"+age+"␣ans");
   }
}
```

Dans la suite, tous les attributs de la classe *Personne* seront privés et toutes les méthodes seront publiques. On peut avoir des méthodes privées ; dans ce cas, elles ne sont utilisables que par d'autres méthodes de la même classe. On peut théoriquement disposer d'attributs publiques, mais ils sont fortement déconseillés.

1.2 Utilisation de la classe Personne

La classe proposée vous permet d'instancier des objets de type *Personne* et appliquer sur ces objets les deux méthodes publiques *initialiser* et *afficher*.

1.2.1 Méthodologie

En Java pour déclarer un objet nous utilisons la déclaration suivante :

```
Personne fille;
```

Cette déclaration est tout à fait correcte. Mais, contrairement à la déclaration d'une variable de type primitif, cette déclaration ne réserve pas d'emplacement mémoire pour l'objet *fille*. L'emplacement pour l'objet sera alloué sur une demande explicite par le biais de l'opérateur *new*.

```
fille=new Personne(); // création d'un objet de type Personne
            // et place sa référence dans fille
```

Pour l'instant, les attributs nom, prenom et age n'ont apparemment reçu aucune valeur (on verra plus tard qu'en réalité, ils ont été initialisé par default à une valeur 0 pour age et null pour nom et prenom). Une fois qu'une référence à un objet est convenablement faite, n'importe qu'elle méthode peuvent être appliquée à l'objet correspondant. Par exemple, nous pouvons appliquer la méthode *initialiser* à l'objet référencé par *fille* comme suit :

```
// appelle la méthode initialiser
fille.initialiser("Toumi","Maria-Lyna",5);
```

Si nous ignorons le préfixe *fille*, cet appel est similaire aux appels classiques des fonctions tels qu'on le rencontre dans la plupart de langages informatique. Mais, il faut faire attention à ce préfixe, c'est lui qui va préciser sur quel objet cette méthode doit

être appliquée. Ainsi, l'instruction nom=n de la méthode *initialiser* affectera la valeur reçue par *n* (dans notre cas "Toumi") dans l'attribut nom de l'objet *fille*.

1.2.2 Exemple

Nous vous proposons d'utiliser la classe *Personne* depuis n'importe quelle méthode d'une autre classe, ou depuis la méthode *main*. La méthode *main* est une méthode statique. A priori, nous pourrions faire de *main* une méthode de la classe *Personne*. Mais la méthodologie serait trop particulière dans ce cas, nous préférons dans notre exemple que la méthode *main* appartient à une autre classe. nous proposons ci-dessous un exemple complet d'une classe nommée *Application* contient (seulement) la méthode *main* et qui utilise la classe *Personne*

```
public class Application
{
  public static void main (String args[])
    {
           Personne fille;
           fille = new Personne();
           fille.initialiser("Toumi","Maria-Lyna", 5);
           fille.afficher();
           Personne fils = new Personne();
           fils.initialiser("Toumi", "Ahmed-Yahia", 2);
           fils.afficher() ;
        }
}
```

résultats :

```
Je suis Toumi Maria-Lyna Age: 5 ans
Je suis Toumi Ahmed-Yahia Age: 2 ans
```

1.3 Constructeur

Dans la classe *Personne* de la section précédente, nous avons vu qu'il est nécessaire de recourir à la méthode *initialiser* pour attribuer des valeurs à un objet. Une telle méthodologie permet à l'utilisateur d'un objet de faire l'appel voulu à un moment donné. En effet, la notion de constructeur, nous permettrons d'automatiser le mécanisme d'initialisation d'un objet lors de la création. Un constructeur est une méthode spéciale, qui n'a aucune valeur de retour, le nom de cette méthode est similaire au nom de sa classe. Un constructeur peut disposer d'un ou plusieurs arguments (éventuellement d'aucun).

1.3.1 Exemple

Considérons la classe *Personne* présentée précédemment, nous procédons à une modification de la méthode *initialiser* en constructeur en la nommant *Personne*. La nouvelle classe devient : Maintenant, comment utiliser la nouvelle classe *Personne* ? en premier lieu nous vous proposons l'instruction suivante :

```
Personne p=new Personne() ;
```

cette proposition n'est plus valide ; elle serait directement refusée par le compilateur. En effet, à partir du moment où un constructeur est défini dans la classe, il n'est plus

```
public class Personne
{
  private String nom ;
  private String prenom ;
  private int age;
  public Personne(String n, String p, int a)
   {
     nom=n ;
     prenom=p ;
     age=a;
   }
  public void afficher()
   {
     System.out.print ("Je␣suis␣"+nom +"␣"+ prenom);
     System.out.println("␣age:␣"+age+"␣ans");
   }
}
```

question de créer un objet avec le constructeur par default, puisque notre nouveau constructeur a besoin de trois arguments qui doivent être obligatoirement fournis lors de la création de l'objet, par exemple :

```
Personne fille =new Personne("Toumi","Maria-Lyna", 5);
```

La nouvelle classe *Application* sera comme suite :

```
public class Application
{
  public static void main (String args[])
    {
          Personne fille;
          fille = new Personne("Toumi","Maria-Lyna", 5);
          fille.afficher();
          Personne fils = new Personne("Toumi", "Ahmed-Yahia", 2);
          fils.afficher();
       }
}
```

Résultats :

```
Je suis Toumi Maria-Lyna Age: 5 ans
Je suis Toumi Ahmed-Yahia Age: 2 ans
```

1.3.2 Règles sur constructeurs

1. Par nature, un constructeur ne doit fournir aucune valeur de retour, même s'il sera signalé avec *void*, il provoque une erreur de compilation :

   ```
   class Exemple
   { // erreur de compilation void
     public void Exemple ()
      {
      }
   }
   ```

2. Si une classe n'a aucun constructeur (le cas de notre première proposition de la classe Personne). L'instanciation des objets est faite par le constructeur par default (sans arguments) :

```
Personne a=new Personne();
   // OK si aucun constructeur n'a été déclaré
```

Mais dès qu'une classe possède au moins un constructeur, le constructeur par default ne peut être jamais utilisé, comme le montre l'exemple suivant :

```
class Exemple
{
   public Exemple(int a) { ..... }
     // constructeur qui avec un argument de type int
}
.....
Exemple e1=new Exemple(5); // OK
Exemple e2=new Exemple(); // erreur
```

Il est à noter que l'utilisation d'un constructeur par default ne se distingue pas de celui du constructeur personnalisé par l'utilisateur. L'instruction suivante est acceptée :

```
C c=new C() ;
```

cela signifie simplement que :
— soit C ne dispose d'aucun constructeur personnalisé,
— soit C dispose d'un constructeur par default.

3. Jamais un constructeur n'est appelé directement depuis une autre méthode. Par exemple, si la classe *Personne* dispose d'un constructeur à deux arguments de type *String* et un argument de type *int* :

```
Personne ami = new Personne("Imlouli","Lakhdar", 45);
.....
ami.Personne("Zebar","Samir", 45);// interdit
```

4. Nous pouvons appeler un autre constructeur au sein du constructeur de la même classe.

5. Lorsque un constructeur est déclaré *privé* (*private*), il ne pourra pas être appeler de l'extérieur de la classe, c'est-à-dire, qu'il ne pourra pas être utilisé pour créer des objets :

```
class B
{
  private B() { ..... } // constructeur privé sans arguments
   .....
}
.....
B b=new B(); // erreur : le constructeur correspondant B() est privé
```

1.4 Surdéfinition de constructeur

Dans cette section nous introduisons le principe de la surdéfinition (ou bien de la surcharge dans le langage *C*) des méthodes. Sans même en avoir conscience, nous

sommes en présence d'un tel mécanisme dans des expressions arithmétiques telles que $a + b$: la signification du symbole + dépend du type des variables a et b.

En Java, cette possibilité de surdéfinition s'applique aux méthodes d'une classe. Plusieurs méthodes peuvent porter le même nom, la seule différence est dans le nombre et le type d'arguments, ça veut dire la signature de la méthode. Cela permet au compilateur d'effectuer son choix.

1.4.1 Exemple introductif

Considérons l'exemple suivant, dans lequel nous avons doté la classe *Personne* de trois constructeurs :

— le premier à deux arguments de type *String*,
— le deuxième à un seul argument de type *int*,
— le troisième à trois arguments de type de type *String* et *int*,

```
class Personne
{
  String nom, prenom;
  int age;
  // constructeur
  public Personne (String n, String p)
   {
     nom=n ;
     prenom=p ;
   }
  // constructeur 2
  public Personne (int a)
   {
     age=a;
   }
  // constructeur 3
  public Personne (String n, String p, int a)
   {
     nom=n ;
     prenom=p ;
     age=a;
   }
}
public class Application
{
  public static void main (String args[])
    {
      // appel du constructeur 3 (String, String,int)
          Personne moi = new Personne("Toumi","Lyazid", 45);
      // appel du constructeur 1 (String, String)
          Personne moi = new Personne("Toumi","Lyazid");
      // appel du constructeur 2 (int)
          Personne moi = new Personne(45) ;
    }
}
```

1.4.2 Surdéfinition et droits d'accès

Nous savons qu'une méthode pouvait être publique ou privée. Dans tous les cas, elle peut être surdéfinie. Cependant, les méthodes privées ne sont pas accessibles en dehors de la classe. Dans ces conditions et suivant son emplacement, un même appel peut conduire à l'appel d'une méthode différente.

```
public class Accessibilite
{
  public static void main (String args[])
    {
          A a=new A();
          a.g();
          System.out.println ("␣dans␣main");
          int n=42 ; float x=23.34f;
          a.f(n); a.f(x);
        }
}
class A
{
  public void f(float x)
   {
      System.out.println ("f(float)␣x=" + x );
   }
  private void f(int n)
   {
          System.out.println ("f(int)␣n=" + n);
   }
  public void g()
   {
      int n=10 ; float x=22.5f;
          System.out.println ("␣dans␣g");
          f(n); f(x);
   }
}
```

```
dans g
f(int) n= 10
f(float) x= 22.5
dans main
f(float) x= 42.0
f(float) x= 23.34
```

Dans *main*, l'appel $a.f(n)$ provoque l'appel de la méthode $f(float)$ de la classe A, car c'est la seule qui sera acceptée (f(int) est privée). En revanche, pour l'appel comparable $f(n)$ effectué au sein de la méthode g de la classe A, les deux méthodes f seront acceptées ; c'est donc $f(int)$ qui est utilisée.

1.5 Attributs et méthodes statiques

En Java, nous avons la possibilité de définir des attributs communs entre les objets de la même classe, c-à-d l'existence d'un seul exemplaire pour tous les objets la même

classe. Il s'agit ici d'une ressource partagée entre tous les objets instanciés de la même classe, nous parlons alors d'un attribut statique. De même, nous pouvons définir des méthodes statiques qui peuvent être appelées indépendamment de tout les autres objets de la classe (c'est le cas de la méthode main).

1.5.1 Attributs statiques

Présentation

Soit la classe suivante (nous ne préoccuperons pas des droits d'accès des attributs n et y) :

```
class A
{
  int n ;
  float y ;
}
```

Chaque objet de type A possède ses propres attributs n et x. Par exemple :

```
A a1=new A(), a2=new A();
```

Mais Java vous permet de définir des attributs statiques qui n'existent qu'en seul exemplaire, quelque soit le nombre d'objets créés à partir de la même classe. Il vous suffit pour les déclarer d'utiliser le mot clé *static*. Par exemple :

```
class B
{
  static int n ;
  float y ;
}
B a1=new B(), a2=new B() ;
```

Les notations a1.n et a2.n désignent donc le même attribut. En effet, cet attribut existe indépendamment de tout les objets de sa classe. Il est possible (et même préférable) de s'y référer directement. Par exemple :

```
B.n // attribut (statique) n de la classe B
```

Bien entendu, ces trois notations (a1.n, a2.n, B.n) ne seront utilisables que pour un attribut publique. Il sera possible de prévoir des attributs statiques privés, mais l'accès ne pourra alors se faire que par le biais de méthodes statiques (l'exemple du main).

Exemple

Pour assurer une bonne compréhension, nous vous proposons un exemple complet qui utilise la classe *Instances* contient un attribut statique privé *nombre*, utiliser pour compter le nombre d'objets de type *Instances* créés. Sa valeur est incrémentée de 1 à chaque appel du constructeur. Nous nous contentons pour l'instant d'afficher la valeur de *nombre* à chaque création d'un nouvel objet.

```
class Instances
{
  private static long nombre=0;
  public Instances()
   {
     System.out.print ("++ creation objet Instances ; ");
         nombre ++;
         System.out.println ("il y en a maintenant "+nombre);
   }
}
public class Application
{
  public static void main (String args[])
   {
         Instances a;
         System.out.println ("Etape 1:");
         a=new Instances();
         System.out.println ("Etape 2:");
         Instances b;
         System.out.println ("Etape 3:");
         b=new Instances();
         Instances c=new Instances();
         System.out.println ("Etape 4.");
   }
}
```

```
Etape 1:
++ creation objet Instances ; il y en a maintenant 1
Etape 2:
Etape 3:
++ creation objet Instances ; il y en a maintenant 2
++ creation objet Instances ; il y en a maintenant 3
Etape 4.
```

1.5.2 Méthodes statique

Généralités

Nous avons vu précédemment comment définir un attribut statique, cet attribut n'existent qu'en un seul exemplaire, indépendamment de tout les objets de la même classe. De manière analogue, on peut imaginer que certaines méthodes statique aient un rôle indépendant d'un quelconque objet. Ce serait notamment le cas d'une méthode se contentant d'agir sur des attributs statiques ou de les utiliser.

Nous pouvons toujours appelé une telle méthode en la faisant porter artificiellement sur un objet de la classe (alors que la référence à un tel objet n'est pas utile). Ici encore, Java vous permet de définir une méthode statique en la déclarant avec le mot-clé *static*. L'appel d'une telle méthode ne nécessite que le nom de la classe correspondante.

Bien entendu, une méthode statique ne pourra en aucun cas agir sur des attributs usuels (non statiques) puisque, par nature, elle n'est liée à aucun objet en particulier.

Par exemple :

```
class A
{
    .....
    // attribut
    private float x ;
    // attribut statique
    private static int n ;
    .....
    // méthode statique
    public static void f()
    {
      // ici, on ne peut pas accéder à x
      .....
      // mais on peut accéder à l'attribut statique n
      .....
    }
}
.....
A a ;
// appelle la méthode statique f de la classe A
A.f() ;
// reste autorisé, mais déconseillé
a.f() ;
```

Exemple

Voici un exemple complet sur l'emploi d'une méthode statique. Il s'agit de l'exemple précédant, mais nous avons introduit une méthode statique nommée nombreObjet affichant simplement le nombre d'objets crées à partir de la classe.

```
class Instances
{
 private static long nombre=0;
 public Instances()
  {
     System.out.print ("++ creation objet Instances ; ");
         nombre ++;
         System.out.println ("il y en a maintenant " + nombre);
  }
 public static long nObj()
  {
    return nombre;
  }
}
public class Application
{
  public static void main (String args[])
   {
        Instances a;
        System.out.println ("Etape 1 : nb objets " + Instances.nObj());
        a=new Instances();
        System.out.println ("Etape 2 : nb objets " + Instances.nObj());
        Instances b;
        System.out.println ("Etape 3 : nb objets " + Instances.nObj());
        b=new Instances();
        Instances c=new Instances();
        System.out.println ("Etape 4 : nb objets " + Instances.nObj());
   }
}
Etape 1 : nb objets 0
++ creation objet Instances; il y en a maintenant 1
Etape 2 : nb objets 1
Etape 3 : nb objets 1
++ creation objet Instances; il y en a maintenant 2
++ creation objet Instances; il y en a maintenant 3
Etape 4 : nb objets 3
```

1.6 Exercices

1.6.1 Création et utilisation d'une classe

Réaliser une classe *Mobile* permettant de représenter un téléphone mobile. Chaque mobile sera caractérisé par sa marque, son modèle et son système (de type String) et son prix (de type double). La classe devra contenir :

- un constructeur recevant en arguments la marque, le modèle, le système et le prix d'un téléphone mobile,
- une méthode *afficher* toutes les informations du mobile,
- une méthode *revendre* effectuant une modification du prix par la valeur de son argument.

Écrire une classe *Application* utilise la classe *Mobile* pour créer le *Iphone X*, et afficher toutes les caractéristiques de ce téléphone mobile. Afficher à nouveau les caractéristiques de ce téléphone après une revente.

1.6.2 Initialisation d'un objet

Que fournit l'exécution de la classe Application ci-dessous ?

```
class Nombre
{
private int n=2;
private int k;
public Nombre(int limit)
 { n *= limit;
   n += k;
 }
public void afficher()
 { System.out.println("n="+n+ "k="+k) ;
 }
}
public class Application
{ public static void main(String args[])
  { Nombre n=new Nombre(5);
    n.afficher();
  }
}
```

1.6.3 Affectation et comparaison d'objets

Que fournit l'exécution de la classe Application ci-dessous ?

```
class NombreEntier
{
  private int e;
  public NombreEntier(int n) {e=n;}
  public void decaler(int d) {e+=d;}
  public void afficher() {System.out.println(e);}
}

public class Application
{
public static void main (String args[])
{
  NombreEntier e1 = new NombreEntier(22);
  System.out.print("e1=␣");
  e1.afficher();
  NombreEntier e2 = new NombreEntier(5);
  System.out.print("e2=␣");
  e2.afficher();
  e2.decaler(17);
  System.out.print("e2␣=␣");
  e2.afficher();
  System.out.println("e1==e2␣est␣"+(e1==e2));
  e1=e2; e2.decaler(12);
  System.out.print ("e2=␣"); e2.afficher();
  System.out.print ("e1=␣"); e1.afficher();
  System.out.println ("e1==e2␣est␣"+(e1==e2));
}
}
```

1.6.4 Méthodes d'accès aux attributs privés

Soit les classes Mobile et Application ci-dessous :

```
class Mobile
{
 private String marque;
 private String modele;
 private String systeme;
 private double prix;
 public Mobile (String ma, String mo, String s, douple p)
 { marque=ma;
   modele=mo;
   systeme=s;
   prix=p;
 }
 public void revendre(double marge) {prix +=marge;}
 public void afficher()
 { System.out.println ("Mobile("+marque+"("+modele+")"+systeme+":"+prix);
 }
}
public class Apllication
{
  public static void main (String args[])
   {
    Mobile iphonex;
    iphonex= new Mobile("Apple","Iphone␣X","IOS","200000");
    iphonex.afficher() ;
    iphonex.revendre(10000); iphonex.afficher();
    galaxys9= new Mobile("Samsung","Galaxy␣S9","Android","150000");
   }
}
```

Modifier la définition de la classe *Mobile* en supprimant la méthode *afficher* et en introduisant quatre méthodes nommées getmarque, getModele, getSysteme et getPrix fournissant respectivement la marque, le modèle, le système et le prix d'un mobile. Adapter la classe Application pour quelle prendre en charge les nouvelles modifications.

1.6.5 Casting

On suppose qu'on dispose de la classe A ainsi définie :

```
class Casting
{
 void fx (int n, float x) {}
 void gx (byte b) {}
}
```

Soit la classe Application ci-dessous, donner le résulta de chaque appels de fx et de gx :

```
public class Apllication
{
  public static void main (String args[])
   {
     Casting c; int n; byte b; float f; double d;
         c.fx(n,f);
         c.fx(b+3,f);
         c.fx(b,f);
         c.fx(n,d);
         c.fx(n,(float)d);
         c.fx(n,2*f);
         c.fx(n+5,f+0.5);
         c.gx(b);
         c.gx(b+1);
         c.gx(b++);
         c.gx(3);
    }
}
```

1.6.6 Attributs et méthodes de classe

Trouver les erreurs dans la définition dans les classes ci-dessous ?

```
class Exemple
{
 static private final double d= 0.1;
 private int i;
 static int fx(int n)
  {
   i=n;
  }
 void gx(int n)
  {
   i=n;
   d=n;
  }
}
public class Application
{
 public static void main (String[] args)
 { Exemple e1=new Exemple(); int ii=5;
  e1.gx(ii);
  e1.fx(ii);
  Exemple.fx(ii);
  fx(ii);
 }
}
```

1.6.7 Attributs et méthodes de classe(2)

Réaliser une classe *Compte* qui permet des créer des comptes bancaires et d'attribuer un numéro unique à chaque nouveau compte bancaire créé. On ne cherchera pas à réutiliser les numéros de comptes détruits. Notre classe est dotée uniquement d'un constructeur, d'une méthode getIdentificateur fournissant le numéro attribué au compte et d'une méthode getIdentificateurMax fournissant le numéro du dernier compte créé. Écrire les classes Compte et Application.

1.6.8 Surdéfinition de méthodes

Trouver les erreurs figurant dans la classe ci-dessous ?

```
class SurDefinition
{
 public void fx(int i){}
 public int fx(int ii){}
 public void gx(float f){}
 public void gx(final double d){}
 public void hx(long l){}
 public int hx(final long ll){}
}
```

1.6.9 Surdéfinition de méthodes(2)

Soit la définition de la classe Definition et Application ci-dessous :

```
class
{
 public void fx(byte b){System.out.println("Je␣suis␣fx(byte)");}
 public void fx(int n){System.out.println("Je␣suis␣fx(int)");}
 public void fx(float x){System.out.println("Je␣suis␣fx(float)");}
 public void fx(double y){System.out.println("Je␣suis␣fx(double)")}
}
public class Application
{
 public static void main (String[] args)
 {
    Definition d; byte b; short s; int i; long l; float f; double d;
    d.fx(b);
        d.fx(s);
        d.fx(l);
        d.fx(f);
        d.fx(d);
        d.fx(2.*f);
        d.fx(b+1);
        d.fx(b++);
 }
}
```

Quelles sont les méthodes appelées par les instructions dans la méthodes *main* ?

Chapitre 2

Tableaux

Les tableaux dans n'importe quel langage de programmation désigne un ensemble d'éléments qui ont le même type et qui ont un nom unique, chacun des éléments du tableau est repéré par un indice repérant ça position au sein de l'ensemble. Java permet de manipuler des tableaux mais différemment des autres langages. En particulier, les tableaux sont considérés comme des objets et les tableaux multidimensionnels sont obtenus par composition de tableaux.

2.1 Déclaration et création d'un tableaux

soit la déclaration suivante :

```
int t[] ;
```

La déclaration précédente précise que t est une référence à un tableau d'entiers. Nous constatons qu'aucune dimension n'est mentionné dans cette déclaration et, pour l'instant, aucune valeur n'a été attribuée à t. Cette déclaration est en effet très proche de celle de la référence à un objet d'une classe.

La création d'un tableau est similaire à la création d'un objet, c'est-à-dire en utilisant l'opérateur *new*. Le type des éléments doit être précisé, ainsi que le nombre de cases (dimension du tableau), comme dans :

```
// t fait référence à un tableau de 5 entiers
t=new int[5] ;
```

L'instruction précédente permet d'allouer l'emplacement nécessaire à un tableau de 5 éléments de type *int* et de placer la référence dans t. Sachant que les éléments d'un tableau sont initialisés par défaut (comme tous les champs d'un objet) à une valeur "null" pour des objets (0 pour les primitives). Le schéma suivant illustre la situation :

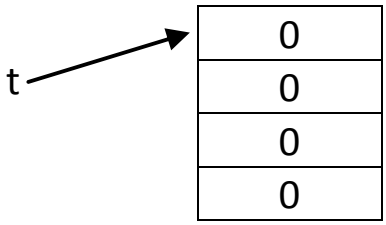
2.1.1 Déclaration de tableaux

La référence à un tableau précise simplement le type des éléments qui compose le tableau. Elle peut prendre deux formes différentes ; Nous prenons l'exemple précédent qu'il sera déclaré comme suit :

```
int t[] ;
```

peut aussi s'écrire :

```
int[] t ;
```



La seule différence entre les deux déclaration est que la première est perceptible lorsque on déclare plusieurs identificateurs dans une même instruction. par exemple,

```
 // t1 et t2 sont des références à des tableaux d'entiers
int[] t1, t2 ;
```

est équivalent à :

```
int t1[], t2[];
```

La première forme permet le mélange de tableaux de type T et de variables de type T :

```
// t1 et t2 sont des tableaux d'entiers, n est entier
int t1[], n, t2[];
```

En Java, les éléments d'un tableau peuvent être d'un type primitif ou d'un type objet. Par exemple, si nous avons défini le type classe *Personne*, ces déclarations sont aussi correctes :

```
 // tp est une référence à un tableau d'objets de type Personne
Personne p[];
// a et b sont des références à des objets de type Personne
Personne moi, p[], autre;
// c est une référence à un tableau d'objets de type Personne
```

Remarque

Une déclaration de tableau ne doit pas préciser de dimensions. Cette instruction sera rejetée à la compilation :

```
// erreur: on ne peut pas indiquer de dimension ici
int t[5];
```

2.1.2 Création d'un tableau

Nous avons vu précédemment, la façon pour allouer un emplacement d'un tableau similaire à la façon utilisée pour un objet à l'aide de l'opérateur *new*.

Création par l'opérateur *new*

L'expression fournie après l'opérateur *new*, n'est calculée qu'au moment de l'exécution du programme. Elle peut être donc différé, contrairement à ce qui se produit dans le cas du langage *C*, où la dimension est fixé lors de la compilation du programme. Voici un exemple suivant :

```
System.out.print ("taille voulue?") ;
int n=5 ;
int t[]=new int [n] ;
```

Le tableau aura une taille fixe une fois sera créé et on pourra pas modifier sa taille. En revanche, comme n'importe quelle référence à un objet, la référence t pourra très bien évoluer au fil de l'exécution et désigner finalement des tableaux de tailles différentes. Enfin, notez que la création d'un tableau de taille null est autorisé (qu'on ne confondra pas avec une référence null). En revanche, l'appel de *new* avec une valeur négative conduira à une exception *NegativeArraySizeException*, nous apprendrons ultérieurement la possibilité d'intercepter une telle exception..

Utilisation d'un initialiseur

Lors de la déclaration d'une référence de tableau, nous pouvons fournir une liste d'expressions entre accolades, tel que dans le cas :

```
int n, p ;
...
int t[]={1, n, n+p, 2*p, 12} ;
```

L'instruction ci-dessus permet de créer un tableau de 5 entiers ayant des valeurs d'expressions mentionnées. Cette dernière remplace les instructions suivantes :

```
int n, p, t[] ;
.....
t=new int[5] ;
t[0]=1 ; t[1]=n ; t[2]=n+p ; t[3]=2*p ; t[4]=12 ;
```

Notez que Java utilise un nombre d'expressions figurant dans l'initialiseur pour en déduire la taille d'un tableau à créer. Notez aussi que ces expressions n'ont pas besoin d'être des expressions constantes ; il suffit simplement qu'elles soient calculables au moment où l'on exécute l'opérateur *new*.

2.2 Utilisation d'un tableau

Deux façon sont utilisées en Java pour utiliser un tableau :
— en accédant individuellement à chacun de ses éléments,
— en accédant globalement à l'ensemble du tableau.

2.2.1 Accès individuel aux éléments d'un tableau

On peut manipuler un élément de tableau comme nous le ferons avec n'importe quelle variable ou n'importe quel objet du type de ses éléments. On désigne un élément particulier en plaçant entre crochets, à la suite du nom du tableau, une expression entière nommée indice indiquant sa position. Le premier élément correspond à l'indice 0 (et non 1).

```
int t[]=new int[5];
// place la valeur 15 dans le premier élément du tableau t
t[0]=15;
// incrémente de 1 le troisième élément de t
t[2]++;
```

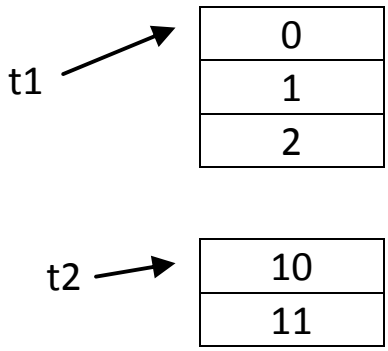
Lorsque la valeur d'un indice sera négative ou trop grande par rapport à la taille du tableau lors de l'exécution, une erreur d'exécution sera obtenue. Plus précisément, il y a un déclenchement d'une exception de type *ArrayIndexOutOfBoundsException*. Si nous utiliserons pas d'exception, un arrêt imminent de l'exécution du programme sera fait ; en fenêtre console s'affichera un message d'erreur.

2.2.2 Affectation de tableaux

Nous avons montré dans le paragraphe précédent comment accéder d'une manière individuelle à chacun des éléments d'un tableau existant. Java permet aussi une manipulation globale des tableaux, par le biais d'affectations de leurs références. soit les instructions ci-dessous qui créent deux tableaux d'entiers t1 et t2 :

```
int[] t1=new int[3];
for(int i=0;i<3;i++) t1[i]=i;
int[] t2=new int[2];
for(int i=0;i<2;i++) t2[i]=10+i;
```

La situation engendrer par les instructions précédentes est schématisée comme suit : Après l'exécution de l'affectation suivante :




```
// la référence contenue dans t2 est recopiée dans t1
t1=t2 ;
```

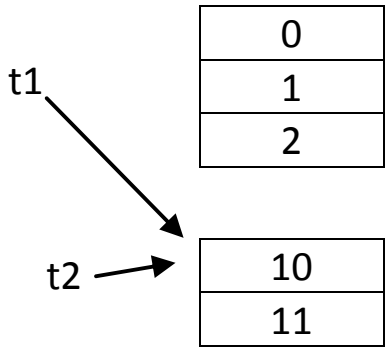
La situation suivante est engendrée : A partir de l'instruction précédente, t1 et t2 désignent le même tableau. Par exemple :

```
t1[1]=5 ;
System.out.println (t2[1]) ;
```

Le résultat obtenu est 5 mais pas 10. Si l'objet qui constitue le tableau de trois entiers anciennement désigné par $t1$ n'est plus référencé, par ailleurs il deviendra candidat au ramasse-miettes pour un nettoyage imminent. Il est très important de noter que l'affectation de références de tableaux n'entraîne aucune recopie des valeurs des éléments du tableau. On retrouve exactement le même phénomène que pour l'affectation d'objets.

2.2.3 Taille d'un tableau : *length*

La déclaration d'une référence de tableau ne précise aucune taille et nous avons vu que cette dernière peut évoluer au fil de l'exécution d'un programme. L'attribut *length*



permet de connaître le nombre d'éléments d'un tableau.

```
int t[]=new int[5] ;
// affiche 5
System.out.println ("taille␣de␣t␣:␣" + t.length) ;
t=new int[3] ;
 // affiche 3
System.out.println ("taille␣de␣t␣:␣" + t.length) ;
```

2.2.4 Exemple d'un tableau d'objets

Nous avons déjà dit que les éléments d'un tableau peuvent être de type quelconque, et pas seulement de type primitif. Voici un exemple de programme utilisant un tableau d'objets de type Personne :

```
public class Application
{
 public static void main (String args[])
  {
    Personne [] c =new Personne[2] ;
    c[0]= new Personne("Toumi","Lyazid",45);
    c[1]= new Personne("Imlouli","Lakhdar",78);
     for (int i=0 ; i<c.length ; i++) c[i].impression() ;
  }
}
public class Personne
{
private String nom, prenom ;
private int age;
public Personne (String n, String p, int a)
 {
  nom=n ;
  prenom=p ;
  age=a;
 }
public void affiche ()
 {
  System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans") ;
 }
}
```

```
Je suis Toumi Lyazid Age: 45 ans
Je suis Imlouli Lakhdar Age: 68 ans
```

2.2.5 Cas particulier des tableaux de caractères

Soit la situation suivante :

```
char[] tc ;
int[] ti ;
.....
// on obtient bien les valeurs des caractères de tc
System.out.println (tc) ;
// on n'obtient pas les valeurs des entiers de ti
System.out.println (ti) ;
```

Le premier affichage est satisfaisant, le second n'est pas. En effet,la méthode println (même choses pour print) est surdéfinie pour des tableaux de caractères. Elle n'est pas surdéfinie pour les autres tableaux, de sorte que, dans le second cas, il y a un appel de la méthode toString de la classe tableau correspondante. Qui plus est, l'instruction suivante :

```
System.out.println ("tc␣" + tc) ;
```

ne donne pas de résultat satisfaisant car la présence de la chaîne entraînera la conversion de tc en chaîne.

2.3 Les tableaux multidimensionnels

La majorité des langages disposent de la notion de tableau multidimensionnel. Par exemple, un tableau bidimensionnel permet de représenter une matrice mathématique. Java ne dispose pas cette notion. Néanmoins, il permet de la simuler en créant des tableaux de tableaux, c'est-à-dire, des tableaux dont les éléments sont eux-mêmes des tableaux. Nous verrons par la suite, cette possibilité s'avère en fait plus riche que celle de tableaux multidimensionnel offerte par les autres langages. Elle permet notamment de créer des tableaux irréguliers, c'est-à-dire dans lesquels les différentes lignes pourront être de taille différente. Bien entendu, on pourra toujours se contenter du cas particulier dans lequel toutes les lignes auront la même taille et, donc, manipuler l'équivalent des tableaux bidimensionnel classiques.

2.3.1 Présentation générale

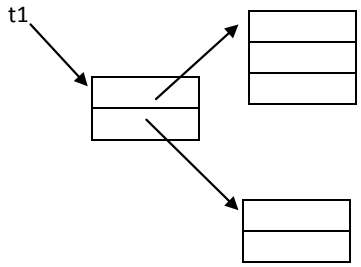
Ces trois déclarations sont équivalentes :

```
int t1 [][] ;
int [][] t1 ;
```

Les déclarations précédentes montre que t1 est une référence à un tableau, dans lequel chaque élément a sa propre référence à un tableau d'entiers. Soit la déclaration suivante :

```
int t1 [][]={new int[3], new int[2]};
```

L'initialiseur de t1 comporte deux éléments dont l'évaluation doit créer un tableau de 3 entiers et un tableau de 2 entiers. On aboutit à cette situation (les éléments des tableaux d'entiers sont, comme d'habitude, initialisés à 0) : Dans ces conditions, on voit que :



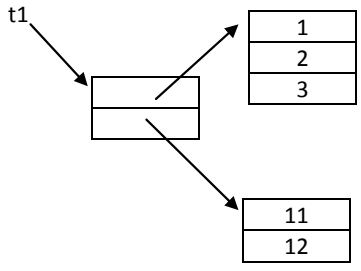
— la notation t1[0] désigne la référence au premier tableau de 3 entiers,
— la notation t1[0][1] désigne le deuxième élément de ce tableau (les indices commencent à 0),
— la notation t1[1] désigne la référence au second tableau de 2 entiers,
— la notation t1[1][i-1] désigne le i_{me} élément de ce tableau,
— l'expression t1.length vaut 2,
— l'expression t1[0].length vaut 3,
— l'expression t1[1].length vaut 2.

2.3.2 Initialisation

Dans le premier exemple, nous avons utilisé un initialiseur pour les deux références à introduire dans le tableau t1 ; autrement dit, nous avons procédé comme pour un tableau à un seul indice. Mais comme on s'y attend, les initialiseurs peuvent tout à fait s'imbriquer, comme dans cet exemple :

```
int t[] []={{1,2,3}, {11,12}};
```

La situation précédente correspond à schéma suivant :



2.4 Exercices

2.4.1 Déclaration et initialisation de tableau

Quelles erreurs ont été commises dans la méthode test ?

```
public void Application ()
{
  public static void main (String[] args)
   {
        int i=20 ;
        final int ii=50;
        int ti[]={1,3,5};
        int ti2[]={i+10,i,i+10};
        int ti3[]={ii-10,ii,ii+11};
        int ti4[];
        ti4={11,13,59};
        float tf1[]={12,22,p,p+21} ;
        float tf2[]={1.85,2.25,52} ;
        double td[]={1,22.5,25.25,29*p};
   }
}
```

2.4.2 Affectation de tableaux

Que se passera-t-il si nous exécutons la méthode test ?

```
public class Tableaux
{
  public void test()
   {
        int ti1[]={11,12,13};
        int ti2[]=new int[4];
        for(int i=0;i<4;i++) ti2[i]=2*i;
          ti2=ti1;
        for(int i=0;i<4;i++) System.out.println(ti2[i]);
   }
}
```

2.4.3 Affectation de tableaux(2)

Quels résultats fournit la méthode test ?

```
public class Tableaux
{
  public void test()
        { char tc1[]={'b','o','n','j','o','u','r'};
          char tc2[]={'h','e','l','l','o'};
          char tc3[]={'x','x','x','x'};
          tc3=tc1; tc1=tc2; tc2=tc3;
          System.out.print("tc1=␣");
          for(int i=0;i<tc1.length;i++) System.out.print(tc1[i]);
          System.out.println();
          System.out.print("tc2=␣");
          for (int i=0;i<tc2.length;i++) System.out.print(tc2[i]);
       System.out.println();
       System.out.print ("tc3␣=␣");
          for (int i=0 ; i<tc3.length ; i++) System.out.print (tc3[i]);
          System.out.println();
    }
}
```

2.4.4 Tableau en argument

Écrire une classe UtiliserTableaux disposant des méthodes statiques suivantes :
— *sommerTab* qui fournit la somme des valeurs d'un tableau de réels (double) de taille quelconque.
— *incrementerTab* qui incrémente d'une valeur donnée toutes les valeurs d'un tableau de réels (double).
— *afficherTab* qui affichage les valeurs d'un tableau de réels.

2.4.5 Tableau en valeur de retour

Écrire une classe UtiliserTableaux2 disposant des méthodes statiques suivantes :
— genererTab qui fournit en retour un tableau des n premiers nombres impairs, la valeur de n étant fournie en argument
— sommerTab qui reçoit en argument deux vecteurs d'entiers de même taille et qui fournit en retour un tableau représentant la somme de ces deux vecteurs.

2.4.6 Tableaux de tableaux

Quels résultats fournit le méthode test ?

```
public class TabDeTableaux
{
  public void test ()
        {
          int[][] m=new int[3][];
          for(int i=0;i<3;i++)
           {
            m[i]=new int[i+1];
                for(int j=0;j<m[i].length;j++)
                        m[i][j]=i+j ;
       }
          for(int i=0;i<3;i++)
           {
             System.out.print ("tableau numero "+i+" = ") ;
             for (int j=0;j<m[i].length;j++)
                 System.out.print (m[i][j]+" ") ;
                 System.out.println();
       }
     }
}
```

Chapitre 3

Héritage

Le concept d'héritage constitue l'un des fondements de la programmation orientée objet. Il est notamment à l'origine des possibilités de la réutilisation des composants logiciels. En effet, il permet de définir une nouvelle classe, dite classe dérivée, à partir d'une classe existante dite classe de base. Cette nouvelle classe hérite les fonctionnalités de la classe de base (attributs et méthodes) qui peuvent être par la suite modifier partiellement ou intégralement, selon les besoin et sans qu'il soit nécessaire de remettre en question la classe de base. Cette technique permet donc de développer de nouveaux outils en se fondant sur d'autres acquis, ce qui justifie le terme d'héritage. Comme on peut s'y attendre, il sera possible de développer à partir d'une classe de base, autant de classes dérivées qu'on le désire. De même, une classe dérivée pourra à son tour servir de classe de base pour une nouvelle classe dérivée. dans ce chapitre nous étalons la notion d'héritage et sa mise en oeuvre en Java. Nous verrons alors ce que devienne les droits d'accès aux attributs et aux méthodes d'une classe dérivée. Puis nous mettons le point sur la construction et l'initialisation des objets dérivés. Nous présentons ensuite les notions d'interfaces dont nous verrons qu'elles remplacent avantageusement l'héritage multiple de certains langages et qu'elle facilitent la tâche de réalisation des classes en imposant le respect d'un certain contrat.

3.1 Notion d'héritage :

dans cette section, nous présentons la façon utilisée en Java pour mettre en oeuvre l'héritage, à partir d'une simple classe sans constructeur. Supposez que nous disposions de la classe *Personne* définie précédemment (pour l'instant, peu importe qu'elle ait été déclarée publique ou non) Si nous voulons des Personnes étudiants, alors nous avons besoin d'une classe Etudiant, destinée à manipuler des personnes étudiants. Une telle classe contient les mêmes fonctionnalités de la classe Personne, auxquelles on pourrait adjoindre, par exemple, une méthode *setSpecialite*, chargée de définir la spécialité de la personne. Dans ces conditions, nous pouvons chercher à définir la classe Etudiant comme une classe dérivée de la classe Personne. Si nous utilisons, la méthode *setSpecialite*, l'attribut specialite, de type String, permet de représenter la spécialité d'une personne, voici comment nous pouvons présenter la définition de la classe Etudiant (ici encore, peu importe qu'elle soit publique ou non) : La mot-clé *extends* précise au compilateur que la classe Etudiant est une classe dérivée de Personne. Nous utilisons la classe *Personne*, nous pouvons déclarer des objets de type Etudiant, par exemple :

```
// Etudiant dérive de Personne
class Etudiant extends Personne
{
  private String specialite;
  public void setSpecialite (String specialite)
   {
    this.specialite=specialite;
   }
}


// e contient une référence à un objet de type Etudiant
Etudiant e ;
 // e2 contient la référence à un objet de
 // type Etudiant créé en utilisant le pseudo constructeur par défaut
Etudiant e2=new Etudiant() ;
```

Un objet de type Etudiant peut alors faire appel :

— aux méthodes publiques de la classe *Etudiant*, ici *setSpecialite* ;

— mais aussi aux méthodes publiques de la classe *Personne* : *initialiser* et *afficher*.

D'une manière générale, un objet d'une classe dérivée accède aux attributs publiques de sa classe de base, exactement comme s'ils étaient définis dans la classe dérivée elle-même.

L'exemple ci-dessous illustre les possibilités (pour l'instant, la classe Etudiant est très rudimentaire ; nous verrons ultérieurement comment la doter par d'autres fonctionnalités indispensables).

```
// classe de base
class Personne
{
private String nom ;
private String prenom ;
private int age;
public void initialiser(String n, String p, int a)
 {
  nom=n ;
  prenom=p ;
  age=a;
 }
public void afficher()
 {
System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans") ;
 }
}
// Etudiant dérive de Personne
class Etudiant extends Personne
{
 private String specialite ;
 public void setSpecialite(String specialite)
  {
   this.specialite=specialite ;
  }
}
// classe utilisant Etudiant
```

```
public class Application
{
  public static void main (String args[])
   {
     Etudiant e=new Etudiant() ;
     e.afficher() ;
     e.initialiser("Toumi","Ahmed", 18) ;
     e.setSpecialite(("Informatique") ;
     e.afficher() ;
     Personne c=new Personne() ; p.initialiser ("Toumi", "Lyazid", 45) ;
     c.afficher() ;
   }
}
```

```
Je suis null null age 0 ans
Je suis Toumi Ahmed age 18 ans
Je suis Toumi Lyazid age 45 ans
```

3.2 Accès d'une classe dérivée aux attributs de sa classe de base

Nous avons vu précédemment, qu'une classe dérivée hérite des attributs et des méthodes de sa classe de base. Mais nous n'avons pas précisé l'usage qu'elle peut en faire. Si nous regardons précisément ce qu'il en est en distinguant les attributs privés des attributs publiques.

3.2.1 Accès aux aux attributs privés par la classe dérivée

Dans l'exemple précédent, nous avons vu comment les attributs publiques dans la classe de base restent des attributs publique dans la classe dérivée. C'est ainsi que nous avons pu appliquer la méthode **initialiser** à un objet de type Etudiant. En revanche, nous n'avons rien dit de la façon dont une méthode de la classe dérivée peut accéder aux attributs de la classe de base. En fait, *Une méthode d'une classe dérivée n'a pas accès aux attributs privés de sa classe de base.* Cette règle peut paraître restrictive. Mais en son absence, il suffirait de créer une classe dérivée pour violer le principe d'encapsulation. Si nous considérons la classe précédente *Etudiant*, elle ne dispose pour l'instant que d'une méthode *afficher*, héritée de la classe *Personne* qui, bien entendu, ne fournit pas la couleur. On peut chercher à la doter d'une nouvelle méthode nommée afficherEtudiant, fournissant à la fois les informations de l'étudiant et sa spécialité. Il ne sera pas possible de procéder ainsi :

```
// méthode affichant les informations et la spécialité de l'étudiant
void afficherEtudiant()
{
// NON : nom ,prenom et age sont des attributs privés
System.out.println ("Je␣suis␣" + nom + "␣" + prénom+"␣age␣"+age+"␣ans") ;
System.out.println ("␣et␣ma␣specialite␣:␣" + specialite) ;
}
```

En effet, la méthode *afficherEtudiant* de la classe Etudiant n'a pas accès aux attributs privés nom, prenom et age de sa classe de base.

3.2.2 Accès aux membres publics

Comme on peut s'y attendre : *Une méthode d'une classe dérivée a l'accès aux attributs publiques de sa classe de base.* Ainsi, pour écrire la méthode *afficherEtudiant*, nous pouvons nous appuyer sur la méthode afficher de la classe *personne* en procédant comme suit :

```
public void afficherEtudiant()
{
  afficher();
  System.out.println ("␣et␣ma␣specialite␣est␣:␣" + specialite);
}
```

Nous notons que l'appel de la méthode *afficher* dans la méthode *afficherEtudiant* est en fait équivalent à :

```
this.afficher();
```

Autrement dit, il applique la méthode *afficher* à l'objet (de type Etudiant) ayant appelé la méthode *afficherEtudiant*. Nous pouvons procéder de même pour définir dans *Etudiant* une nouvelle méthode d'initialisation nommée *initialiserE*, chargée d'attribuer les informations, l'age et la spécialité à un étudiant :

```
public void initialiserE(String nom,String prenom,int age,String specialite)
{
  initialiser (nom, prenom, age);
  this.specialite=specialite;
}
```

3.2.3 Exemple complet

```
public class Personne
{
private String nom;
private String prenom;
private int age;
public Personne (String nom,String prenom,int age)
 {
  this.nom=nom;
  this.prenom=prenom;
  this.age=age;
 }
public void afficher()
 {
   System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
 }
}
class Etudiant extends Personne
{
  private String specialite;

  public void setSpecialite (String specialite)
   {
    this.specialite=specialite;
   }
```

```
  public void afficherEtudiant()
   {
     afficher();
     System.out.println ("␣et␣ma␣specialite␣est␣:␣" + specialite);
   }
  public void initialiserE(String nom,String prenom,int age,String specialite)
   {
     initialiser(nom, prenom, age);
     this.specialite=specialite;
   }
}

public class Application
{
  public static void main (String args[])
   {
     Etudiant e1=new Etudiant();
     e1.initialiser("Toumi", "Maria", 5);
     e1.setSpecialite("Informatique");
     e1.afficher();
     e1.afficherEtudiant();
     Etudiant e2=new Etudiant();
     e2.initialiserE("Toumi", "Yahia", 2, "Medecine");
     e2.afficherEtudiant();
   }
}
```

```
Je suis en Toumi Maria age: 5 ans
Je suis en Toumi Maria age: 5 ans
et ma specialite est: Informatique
Je suis en Toumi Yahia age: 2 ans
et ma specialite est: Medecine
```

3.3 Construction et initialisation des objets dérivés

Nous avons vu précédemment, le choix volontaire des classes sans constructeurs. Mais pratiquement, la plupart des classes les contient. maintenant, nous examinons les différentes situations possibles de constructeurs (présence ou absence de constructeur dans la classe de base et dans la classe dérivée). Puis nous préciserons, comme nous l'avons fait pour les classes simples (non dérivées), la chronologie des différentes opérations d'initialisation (implicites ou explicites) et d'appel des constructeurs.

3.3.1 Appels des constructeurs

Rappelons que dans le cas d'une classe simple (non dérivée), la création d'un objet par *new* entraîne l'appel d'un constructeur avec ou sans arguments. Si aucun constructeur n'est déclaré, on a affaire alors à un pseudo-constructeur par défaut (qui ne fait rien). Voyons ce que deviennent ces règles avec une classe dérivée.

Exemple introductif

Si nous Examinons d'abord un exemple simple dans lequel la classe de base (Personne) et la classe dérivée (Etudiant) disposent toutes les deux d'un constructeur (ce qui correspond à la situation la plus courante en pratique). Pour fixer les idées, supposons que nous utilisions le schéma suivant, dans lequel la classe Personne dispose d'un constructeur à trois arguments et la classe Etudiant d'un constructeur à quatre arguments :

```
class Personne
{
  private String nom, prenom;
  private int age;
   public Personne(String nom, String prenom, int age)
    {
      .....
    }
      .....
}
class Etudiant extends Personne
{
 private String specialite;

  public Etudiant (String nom,String prenom,int age,String specialite)
   {
    .....
   }
    .....
}
```

Tout d'abord, il faut savoir que : *En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.* S'il est nécessaire d'initialiser certains attributs de la classe de base et qu'ils sont convenablement encapsulés, il faudra disposer de fonctions qui utilisent un constructeur de la classe de base. Ainsi, le constructeur de Etudiant pourrait :

- initialiser l'attribut specialite (accessible, car membre de Etudiant),
- appeler le constructeur de Personne pour initialiser les attributs nom, prenom et age.

Pour illustrer cette situation, il est impératif de respecter une règle imposée par Java : *Si un constructeur d'une classe dérivée appelle un constructeur d'une classe de base, ce dernier doit obligatoirement s'agir de la première instruction du constructeur et désigner par le mot-clé super.* Dans notre cas, voici ce que pourrait être cette instruction :

```
// appel d'un constructeur de la classe de base, auquel
// on fournit en arguments les valeurs de nom,prenom et age
super(nom,prenom,age) ;
```

D'où notre constructeur de Etudiant :

```
public Etudiant(String nom,String prenom,int age,String specialite)
{
// obligatoirement comme première instruction
super(nom,prenom,age) ;
this.specialite=specialite;
}
```

La classe de base ne possède aucun constructeur

Il reste possible d'appeler le constructeur par défaut dans la classe dérivée, comme dans :

```
class A
{
..... // aucun constructeur
}
class B extends A
{
// constructeur de B
  public B(...)
   {
// appelle ici le pseudo constructeur par défaut de A
     super() ;
     .....
   }
}
```

L'appel do mot-clé *super()* dans cette situation est superflu, mais il ne nuit pas. En fait, cette possibilité s'avère pratique lorsque on définit une classe dérivée sans connaître les détails de la classe de base. On peut ainsi s'assurer qu'un constructeur sans argument de la classe de base sera toujours appelé et, si cette dernière est bien conçue, que la partie de B héritée de A sera donc convenablement initialisée. Bien entendu, il reste permis de ne doter la classe B d'aucun constructeur. Nous verrons qu'il y aura alors appel d'un constructeur par défaut de A ; comme ce dernier ne fait rien, au bout du compte, il ne se passera rien de particulier !

La classe dérivée ne possède aucun constructeur

Dans cette situation las classe dérivée ne possède pas de constructeur, alors il n'est bien sûr plus question de prévoir un appel explicite (par super) d'un quelconque constructeur de la classe de base. On sait déjà que, dans ce cas, tout se passe comme s'il y avait appel d'un constructeur par défaut sans argument. Dans le cas d'une classe simple, ce constructeur par défaut ne faisait rien (nous l'avons d'ailleurs qualifié de "pseudo-constructeur"). Dans le cas d'une classe dérivée, il est prévu qu'il appelle un constructeur sans argument de la classe de base. On va retrouver ici les règles correspondant à la création d'un objet sans argument, ce qui signifie que la classe de base devra :

- soit posséder un constructeur public sans argument, lequel sera alors appelé,
- soit ne posséder aucun constructeur ; il y aura appel du pseudo-constructeur par défaut.

Voici quelques exemples.

Exemple 1

La construction d'un objet de type B entraîne l'appel du constructeur sans argument de A.

```
class A
{
  // constructeur 1 de A
  public A() { ..... }
  // constructeur 2 de A
  public A (int n) { ..... }
}
class B extends A
{
  ..... // pas de constructeur
}
// construction de B
// appel de constructeur 1 de A
B b=new B() ;
```

Exemple 2

```
class A
{
  // constructeur 2 seulement
  public A(int n) { ..... }
}
class B extends A
{
  ..... // pas de constructeur
}
```

Ici, nous obtenons une erreur de compilation car le constructeur par défaut de B cherche à appeler un constructeur sans argument de A. Comme cette dernière dispose d'au moins un constructeur, il n'est plus question d'utiliser le constructeur par défaut de A.

Exemple 3

```
class A
{
..... // pas de constructeur
}
class B extends A
{
..... // pas de constructeur
}
```

Cet exemple ressemble au précédent, avec cette différence que A ne possède plus de constructeur. Aucun problème ne se pose plus. La création d'un objet de type B entraîne l'appel du constructeur par défaut de B, qui appelle le constructeur par défaut de A.

3.3.2 Initialisation d'un objet dérivé

Nous avons vu jusqu'ici les constructeurs impliqués dans la création d'un objet dérivé. Mais comme nous l'avons déjà signalé précédemment nous utilisons des classes simples, la création d'un objet fait intervenir plusieurs phases :

— allocation mémoire,
— initialisation par défaut des attributs,

— initialisation explicite des attributs,
— exécution des instructions du constructeur.

La généralisation à un objet d'une classe dérivée est assez intuitive. Supposons que : class B extends A ..... La création d'un objet de type B se déroule en 6 étapes.

1. Allocation mémoire pour un objet de type B ; il s'agit bien de l'intégralité de la mémoire nécessaire pour un tel objet, et pas seulement pour les attributs propres à B (c'est-à-dire non hérités de A).
2. Initialisation par défaut de tous les attributs de B (aussi bien ceux hérités de A, que ceux propres à B) aux valeurs "nulles" habituelles.
3. Initialisation explicite, s'il y a lieu, des attributs hérités de A ; éventuellement, exécution des blocs d'initialisation de A.
4. Exécution du corps du constructeur de A.
5. Initialisation explicite, s'il y a lieu, des attributs propres à B ; éventuellement, exécution des blocs d'initialisation de B.
6. Exécution du corps du constructeur de B.

3.4 Redéfinition et surdéfinition

Nous avons déjà vu précédemment la notion de surdéfinition de méthode à l'intérieur d'une même classe. Nous avons vu qu'elle correspondait à des méthodes de même nom, mais de signatures différentes. dans cette section, nous montrons comment cette notion se généralise dans le cadre de l'héritage : *une classe dérivée pourra à son tour surdéfinir une méthode d'une classe ascendante*. Bien entendu, la ou les nouvelles méthodes ne deviendront utilisables que par la classe dérivée ou ses descendantes, mais pas par ses ascendantes. Mais auparavant, nous vous présentons la notion fondamentale de redéfinition d'une méthode. Une classe dérivée peut en effet fournir une nouvelle définition d'une méthode d'une classe ascendante. Cette fois, il s'agira non seulement de méthodes de même nom(comme pour la surdéfinition), mais aussi de même signature et de même type de valeur de retour. Alors que la surdéfinition permet de cumuler plusieurs méthodes de même nom, la redéfinition substitue une méthode à une autre. Avec l'importance de son utilisation, en particulier dans le polymorphisme que nous verrons plus tard, nous vous présenterons d'abord cette notion de redéfinition indépendamment des possibilités de surdéfinition. Nous verrons ensuite comment surdéfinition et redéfinition peuvent intervenir conjointement et nous vous livrerons les règles générales correspondantes.

3.4.1 Notion de redéfinition de méthode

précédemment, nous avons vu qu'un objet d'une classe dérivée peut accéder à toutes les méthodes publiques de sa classe de base. Considérons : L'appel p.afficher() fournit tout naturellement les information de l'objet p de type Personne. L'appel e.afficher() fournit également les informations de l'objet e de type Personne, mais bien entendu, il n'a aucune raison d'en fournir la spécialité. Java nous permet de redéfinir des méthodes. la Redéfinition permet à une classe dérivée de redéfinir une méthode de sa classe de base (mère), en proposant une nouvelle définition. Encore il faut respecter la signature de la méthode (type des arguments), ainsi que le type de la valeur de retour. C'est alors cette nouvelle méthode qui sera appelée sur tout objet de la classe dérivée, en masquant

```
class Personne
{
  private String nom, prenom;
  private int age;
  public void afficher()
   {
   System.out.println ("Je␣suis␣en␣"+nom +"␣"+prenom+"␣age:␣"+age+"␣ans");
   }
}
class Etudiant extends Personne
{
  private String specialite;
  // ici, on suppose qu'aucune méthode ne se nomme afficher
}
Personne p; Etudiant e;
```

en quelque sorte la méthode de la classe de base. Nous pouvons donc définir dans Etudiant une méthode *afficher* reprenant la définition actuelle de *afficherEtudiant*. Si les informations de la classe Personne sont encapsulées et si cette dernière ne dispose pas de méthodes d'accès, nous devrons utiliser dans cette méthode *afficher()* de *Etudiant* la méthode afficher de Personne. Dans ce cas, un petit problème se pose ; en effet, nous pourrions être tentés d'écrire notre nouvelle méthode (en changeant simplement l'en-tête afficherEtudiant() en afficher()) :

```
class Etudiant extends Personne
{
   public void afficher()
    {
       afficher() ;
       System.out.println ("␣et␣ma␣specialite␣est␣" + specialite) ;
    }
  .....
}
```

Or, l'appel de la méthode *afficher()* provoque un appel récursif de la méthode *afficher()* de Etudiant. Il faut donc préciser qu'on souhaite appeler non pas la méthode *afficher* de la classe Etudiant, mais la méthode afficher de sa classe de base Personne. Il suffit pour cela d'utiliser le mot-clé *super*, de la façon suivante :

```
public void afficher()
{
  // appel de la méthode imprimer de la super classe
  super.afficher() ;
  System.out.println("␣et␣ma␣specialite␣est␣" + specialite) ;
}
```

Dans cette situation, l'appel e.afficher() entraînera bien l'appel de afficher de Etudiant, laquelle, comme nous l'espérons, appellera afficher de Personne, avant d'afficher la spécialité de cette personne. Voici un exemple complet de programme illustrant cette possibilité :

```
public class Personne
{
private String nom;
private String prenom;
private int age;
public Personne(String nom,String prenom,int age)
 {
  this.nom=nom;
  this.prenom=prenom;
  this.age=age;
 }
public void afficher()
 {
   System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
 }
}
class Etudiant extends Personne
{
  private String specialite;

  public void Etudiant(String nom,String prenom,int age, String specialite)
   {
     super(nom, prenom, age);
     this.specialite=specialite;
   }
  public void afficher()
   {
     super.afficher();
     System.out.println ("␣et␣ma␣specialite␣est␣:␣" + specialite);
   }
}

public class Application
{
  public static void main (String args[])
    {
      Etudiant e=new Etudiant("Toumi", "Maria", 5);
      e.afficher();
      // ici, il s'agit de imprimer de Etudiant
    }
}
```

```
Je suis en Toumi Maria age: 5 ans
et ma specialite est: Informatique
```

3.4.2 Surdéfinition et héritage

Jusqu'à maintenant, nous n'avions considéré la surdéfinition qu'au sein d'une même classe. En Java, une classe dérivée peut surdéfinir une méthode d'une classe de base (ou, plus généralement, d'une classe ascendante). En voici un exemple :

```
class A
{
  public void f(int n) { ..... }
  .....
}
class B extends A
{
  public void f(float x) { ..... }
  .....
}
A a; B b;
int n; float x;
.....
// appelle f(int) de A
a.f(n);
// erreur de compilation : une seule
// méthode acceptable (f(int) de A
a.f(x);
// et on ne peut pas convertir x de float en int
// appelle f (int) de A
b.f(n);
// appelle f(float) de B
b.f(x);
```

Ici la recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage, jamais en la descendant... C'est pourquoi l'appel a.f(x) ne peut être satisfait, malgré la présence dans B d'une fonction f qui conviendrait.

3.4.3 Utilisation simultanée de surdéfinition et de redéfinition

Surdéfinition et redéfinition peuvent cohabiter. Voyez cet exemple :

```
class A
{
  .....
  public void f(int n) { ..... }
  public void f(float x) { ..... }
}
class B extends A
{
  .....
  // redéfinition de f(int) de A
  public void f(int n) { ..... }
  // surdéfinition de f (de A et de B)
  public void f(double y) { ..... }
}
//
//
```

```
A a; B b;
int n; float x; double y;
.....
// appel de f(int) de A
//(mise en jeu de surdéfinition dans A)
a.f(n);
// appel de f(float) de A
//(mise en jeu de surdéfinition dans A)
a.f(x);
// erreur de compilation
a.f(y) ;
// appel de f(int) de B
//(mise en jeu de redéfinition)
b.f(n) ;
// appel de f(float) de A
//(mise en jeu de surdéfinition dans A et B)
b.f(x) ;
// appel de f(double) de B
//(mise en jeu de surdéfinition dans A et B)
b.f(y) ;
```

3.4.4 Contraintes portant sur la redéfinition

Valeur de retour

Lorsque une méthode est surdéfinit, on n'est pas obligé de respecter le type de la valeur de retour. Cet exemple est légal :

```
class A
{
  .....
  public int f(int n) { .....}
}
class B extends A
{
  .....
  public float f(float x) { ..... }
}
```

En revanche, en cas de redéfinition, Java impose non seulement l'identité des signatures, mais aussi celle du type de la valeur de retour :

```
class A
{
  public int f(int n) { ..... }
}
class B extends A
{
  public float f(int n) { ..... } // erreur
}
```

Dans cette situation, on n'a pas affaire à une surdéfinition de f puisque la signature de la méthode est la même dans A et dans B. Il devrait donc s'agir d'une redéfinition, mais comme les types de retour sont différents, on aboutit à une erreur de compilation.

Les droits d'accès

Considérons cet exemple :

```
class A
{
 public void f(int n) { ..... }
}
class B extends A
{
 // tentative de redéfinition de f de A
 private void f(int n) { ..... }
}
```

Cette proposition sera rejetée par le compilateur. S'il était accepté, un objet de classe A aurait accès à la méthode f, alors qu'un objet de classe dérivée B n'y aurait plus accès. La classe dérivée romprait en quelque sorte le contrat établi par la classe A. C'est pourquoi la redéfinition d'une méthode ne doit pas diminuer les droits d'accès à cette méthode. En revanche, elle peut les augmenter, comme dans cet exemple :

```
class A
{
  private void f(int n) { ..... }
}
class B extends A
{
   // redéfinition de f avec extension
  public void f(int n) { ..... }
} // des droits d'accès
```

Ici, on redéfinit f dans la classe dérivée et, en plus, on la rend accessible à l'extérieur de la classe. On pourrait penser qu'on viole ainsi l'encapsulation des données. En fait, il n'en est rien puisque la méthode f de B n'a pas accès aux membres privés de A. Simplement, tout se passe comme si la classe B était dotée d'une fonctionnalité supplémentaire par rapport à A.

3.4.5 Duplication d'attributs

Bien que cela soit d'un usage peu courant, une classe dérivée peut définir un attribut portant le même nom qu'un attribut d'une classe de base ou d'une classe ascendante. Considérons cette situation, dans laquelle nous avons exceptionnellement prévu des attributs publics :

```
class A{ public int n;}
class B extends A
{ public float n ;
  public void f()
   { // n désigne l'attribut n (float) de B
     n=5.25f ;
     // tandis que super.n désigne
     // le attribut n (int) de la super
     // classe de B
     super.n=3;
   }
}
A a; B b;
```

```
// a.n désigne ici l'attribut n(int) de la classe A
a.n=5;
// b.n désigne ici l'attribut n(float) de la classe B
b.n=3.5f;
```

Il n'y a donc pas de redéfinition de l'attribut n, comme dans le cas de redéfinir une méthode, mais ici c'est une création d'un nouveau attribut qui s'ajoute à l'ancien. Toutefois, alors que les deux attributs peuvent encore être utilisés depuis la classe dérivée, seul l'attribut de la classe dérivée qui n'est pas visible à partir de l'extérieur. Voici un petit programme complet très artificiel illustrant la situation :

```
class A
{ public int n=4; }
class B extends A
{ public float n=4.5f; }
public class DupChamp
{
  public static void main(String[]args)
   { A a=new A(); B b=new B();
     System.out.println ("a.n␣" + a.n) ;
     System.out.println ("b.n␣" + b.n) ;
   }
}


a.n=4
b.n=4.5
```

3.5 Polymorphisme

Le langage Java permet de mettre en oeuvre un puissant concept appelé polymorphisme. Il s'agit d'un concept extrêmement puissant en P.O.O., qui est un concept complémentaire de l'héritage. Le polymorphisme permet de manipuler des objets de types différents. Par exemple, on pourra construire un tableau d'objets, les uns étant de type Personne, les autres étant de type Etudiant (classe héritière de Personne) et appeler la méthode imprimer pour chacun des objets du tableau. Chaque objet réagira en fonction de son propre type. Mais il ne s'agira pas de traiter ainsi n'importe quel objet. Nous montrerons que le polymorphisme exploite la relation d'héritage en appliquant la règle suivante : un étudiant est aussi une personne, on peut donc bien le traiter comme une personne, la réciprocité est bien sûr fausse.

3.5.1 Bases du polymorphisme

Considérons cette situation dans laquelle les classes Personne et Etudiant sont censées disposer chacune d'une méthode afficher(), ainsi que des constructeurs habituels (respectivement à trois et quatre arguments) :

```
class Personne
{
  public Personne (String nom, String prenom, int age) { ..... }
  public void afficher() { ..... }
}
//
```

```
class Etudiant extends Personne
{
  public Etudiant (String nom, String prenom, int age, String specialite)
  public void afficher() { ..... }
}
//Avec ces instructions:
Personne p ;
p=new Personne ("Toumi", "Maria", 5);
```

Mais il se trouve que Java autorise ce genre d'affectation (p étant toujours de type Personne)

```
// p de type Personne contient la référence
p=new Etudiant("Toumi", "Maria", 5, "Informatique");
// à un objet de type Etudiant
```

D'une manière générale, Java permet d'affecter à un objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé. On peut dire qu'on est en présence d'une conversion implicite (légale) d'une référence à un type classe T en une autre référence à un type ascendant de T ; on parle aussi de compatibilité par affectation entre un type classe et un type ascendant. Considérons maintenant ces instructions :

```
Personne p=new Personne ("Toumi", "Maria", 5);
// appelle la méthode afficher de la classe Personne
p.afficher();
p=new Etdiant("Toumi", "Maria", 5, "Informatique");
// appelle la méthode afficher de la classe Etudiant
p.afficher();
```

Dans la dernière instruction, la variable p est de type Personne, alors que l'objet référencé par p est de type Etudiant. L'instruction p.afficher() appelle alors la méthode afficher de la classe Etudiant. Autrement dit, elle se fonde, non pas sur le type de la variable p, mais sur le type effectif de l'objet référencé par p au moment de l'appel (ce type pouvant évoluer au fil de l'exécution). Ce choix d'une méthode au moment de l'exécution (et non plus de la compilation) porte généralement le nom de liaison dynamique. En résumé, le polymorphisme est :

- la compatibilité par affectation entre un type classe et un type ascendant,
- la liaison dynamique des méthodes.

Le polymorphisme permet d'obtenir un comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit.

Exemple 1

```
public class Personne
{
private String nom;
private String prenom;
private int age;
public Personne(String nom,String prenom,int age)
 {
  this.nom=nom;
  this.prenom=prenom;
  this.age=age;
 }
public void afficher()
 {
   System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
 }
}
class Etudiant extends Personne
{
  private String specialite;

  public void Etudiant(String nom,String prenom,int age, String specialite)
   {
     super(nom, prenom, age);
     this.specialite=specialite;
   }
  public void afficher()
   {
     super.afficher();
     System.out.println ("␣et␣ma␣specialite␣est␣:␣" + specialite);
   }
}
public class Application
{
public static void main (String args[])
   {
   Personne p=new Personne ("Toumi", "Maria", 5);
    // appelle la méthode afficher de la classe Personne
    p.afficher();
    Etudiant e=new Etdiant("Toumi", "Maria", 5, "Informatique");
    p=e;
    // appelle la méthode afficher de la classe Etudiant
    p.afficher();
    p=new Personne ("Toumi", "yahia", 2);
     // appelle la méthode afficher de la classe Personne
    p.afficher();
   }
}
```

```
Je suis en Toumi Maria age: 5 ans
Je suis en Toumi Maria age: 5 ans
et ma specialite est: Informatique
Je suis en Toumi Yahia age: 2 ans
```

Exemple 2

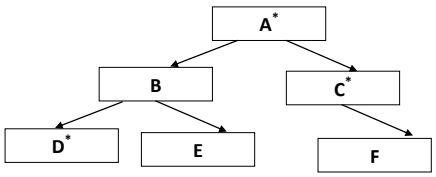
Ce deuxième exemple exploite les possibilités de polymorphisme pour créer un tableau "hétérogène" d'objets, c'est-à-dire dans lequel les éléments peuvent être de type différent.

```
public class Application
{
  public static void main (String args[])
   { Personne[] population new Personne[4] ;
         population[0]=new Personne("Toumi","Maria",5);
         population[1]=new Etudiant("Toumi","Lyna",25,"Informatique");
         population[2]=new Etudiant("Toumi","Ahmed",45,"Medecine");
         population[3]=new Personne("Toumi","Yahia",2);
         for (int i=0;i< population.length;i++) population[i].afficher() ;
   }
}
```

```
Je suis en Toumi Maria age: 5 ans
Je suis en Toumi Lyna age: 25 ans
et ma specialite est: Informatique
Je suis en Toumi Ahmed age: 45 ans
et ma specialite est: Medecine
Je suis en Toumi Yahia age: 2 ans
```

3.5.2 Polymorphisme à plusieurs classes

Nous avons vu précédemment les fondements du polymorphisme, où nous avons considéré que deux classes. Soit la hiérarchie de classes suivante, dans laquelle seules les classes marquées par un astérisque définissent ou redéfinissent la méthode f : Avec



ces déclarations :

```
A a; B b; C c; D d; E e; F f;
les affectations suivantes sont légales:
a=b; a=c; a=d; a=e; a=f;
b=d; b=e;
c=f;
```

En revanche, celles-ci ne le sont pas :

```
b=a; // erreur : A ne descend pas de B
d=c; // erreur : C ne descend pas de D
c=d; // erreur : D ne descend pas de C
```

Voici quelques exemples précisant la méthode f appelée, selon la nature de l'objet effectivement référencé par a (de type A) :

```
a référence un objet de type A : méthode f de A
a référence un objet de type B : méthode f de A
a référence un objet de type C : méthode f de C
a référence un objet de type D : méthode f de D
a référence un objet de type E : méthode f de A
a référence un objet de type F : méthode f de C
```

3.5.3 Polymorphisme, redéfinition et surdéfinition

Le polymorphisme est fondé sur la redéfinition des méthodes. Mais il est aussi possible de surdéfinir une méthode. Cependant, nous n'avions pas tenu compte alors des possibilités de polymorphisme qui peuvent conduire à des situations assez complexes. En voici un exemple :

```
class A
{
  public void f (float x) { ..... }
  .....
}
class B extends A
{
  // redéfinition de f de A
  public void f (float x) { ..... }
  // surdéfinition de f pour A et B
  public void f (int n) { ..... }
  .....
}
A a=new A(...);
B b=new B(...); int n;
// appelle f (float) de A (ce qui est logique)
a.f(n);
// appelle f(int) de B comme on s'y attend
b.f(n);
// a contient une référence sur un objet de type B
a=b;
// appelle f(float) de B et non f(int)
a.f(n);
```

Ainsi, bien que les instructions b.f(n) et a.f(n) appliquent toutes les deux une méthode f à un objet de type B, elles n'appellent pas la même méthode. Voyons plus précisément pourquoi. En présence de l'appel a.f(n), le compilateur recherche la meilleure méthode (règles de surdéfinition) parmi toutes les méthodes de la classe correspondant au type de a (ici A) ou ses ascendantes. Ici, il s'agit de void f(float x) de A. À ce stade donc, la signature de la méthode et son type de retour sont entièrement figés. Lors de l'exécution, on se fonde sur le type de l'objet référencé par a pour rechercher une méthode ayant la signature et le type de retour voulus. On aboutit alors à la méthode void f (float x) de B, et ce malgré la présence (également dans B) d'une méthode qui serait mieux adaptée au type de l'argument effectif. Ainsi, malgré son aspect ligature dynamique, le polymorphisme se fonde sur une signature et un type de retour définis à la compilation (et qui ne seront donc pas remis en question lors de l'exécution).

3.5.4 Règles du polymorphisme en Java

Dans les situations usuelles, le polymorphisme est facile à comprendre et à exploiter. Cependant, nous avons vu que l'abus des possibilités de surdéfinition des méthodes pouvait conduire à des situations complexes. Aussi, nous vous proposons ici de récapituler les différentes règles.

Compatibilité. Il existe une conversion implicite d'une référence à un objet de classe T en une référence à un objet d'une classe ascendante de T (elle peut intervenir aussi bien dans les affectations que dans les arguments effectifs). **Liaison dynamique**. Dans un appel de la forme x.f(...) où x est supposé de classe T, le choix de f est déterminé ainsi :

- **à la compilation :** on détermine dans la classe T ou ses ascendantes la signature de la meilleure méthode f convenant à l'appel, ce qui définit du même coup le type de la valeur de retour.
- **à l'exécution :** on recherche la méthode f de signature et de type de retour voulus, à partir de la classe correspondant au type effectif de l'objet référencé par x (il est obligatoirement de type T ou descendant) ; si cette classe ne comporte pas de méthode appropriée, on remonte dans la hiérarchie jusqu'à ce qu'on en trouve une (au pire, on remontera jusqu'à T).

3.6 Classes et méthodes finales

Le mot-clé *final* vu précédemment pouvait s'appliquer à des variables locales ou à des attributs d'une classe. Il interdit la modification de leur valeur. Ce mot-clé peut aussi s'appliquer à une méthode ou à une classe, mais avec une signification totalement différente : *Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.* Le comportement d'une méthode finale est donc complètement défini et il ne peut plus être remis en cause, sauf si la méthode appelle elle-même une méthode qui n'est pas déclarée final. *Une classe déclarée final ne peut plus être dérivée*, On est ainsi certain que le contrat de la classe sera respecté. On pourrait croire qu'une classe finale est équivalente à une classe non finale dont toutes les méthodes seraient finales. En fait, ce n'est pas vrai car :

- ne pouvant plus être dérivée, une classe finale ne pourra pas se voir ajouter de nouvelles fonctionnalités,
- une classe non finale dont toutes les méthodes sont finales pourra toujours être dérivée, donc se voir ajouter de nouvelles fonctionnalités. Par son caractère parfaitement défini, une méthode finale permet au compilateur :
  - de détecter des anomalies qui, sans cela, n'apparaîtraient que lors de l'exécution,
  - d'optimiser certaines parties de code : appels plus rapides puisque indépendants de l'exécution, mise "en ligne" du code de certaines méthodes...

3.7 Classes abstraites

Une classe abstraite est une classe qui ne permet pas d'instancier des objets. Elle ne peut servir que de classe de base pour une dérivation. Elle se déclare ainsi :

```
abstract class A
{ .....
}
```

Dans une classe abstraite, on peut trouver classiquement des méthodes et des champs, dont héritera toute classe dérivée. Mais on peut aussi trouver des méthodes dites abstraites, c'est à dire dont on ne fournit que la signature et le type de la valeur de retour. Par exemple :

```
abstract class A
{ // f est définie dans A
  public void f() { ..... }
  // g n'est pas définie dans A; on n'en
  // a fourni que l'en tête
  public abstract void g(int n);
}
```

Bien entendu, on pourra déclarer une variable de type A :

```
// OK : a n'est qu'une référence sur un objet de type A ou dérivé
A a;
```

En revanche, toute instanciation d'un objet de type A sera rejetée par le compilateur :

```
// erreur: pas d'instanciation d'objets d'une classe abstraite
a=new A(...) ;
```

En revanche, si on dérive de A une classe B qui définit la méthode abstraite g :

```
class B extends A
{
  // ici, on définit g
  public void g(int n) { ..... }
  .....
}
```

on pourra alors instancier un objet de type B par new B(...) et même affecter sa référence à une variable de type A :

```
A a=new B(...); // OK
```

3.7.1 Quelques règles

1. Une classe qui comporte une ou plusieurs méthodes abstraites, elle est abstraite, et sans même le mot-clé *abstract* devant sa déclaration (ce qui reste quand même vivement conseillé). Ceci est correct :

   ```
   class A
   {
      public abstract void f(); // OK
       .....
   }
   ```

   Malgré tout, A est considérée comme abstraite et une expression telle que new A(...) sera rejetée.

2. Une méthode abstraite doit obligatoirement être déclarée public, ce qui est logique puisque sa vocation est d'être redéfinie dans une classe dérivée.

3. Dans l'en-tête d'une méthode déclarée abstraite, les noms d'arguments muets doivent figurer (bien qu'ils ne servent à rien) :

```
abstract class A
{
 // erreur : nom d'argument (fictif) obligatoire
   public abstract void g(int) ;
}
```

4. Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune). Dans ce cas, elle reste simplement abstraite (il est quant même nécessaire de mentionner abstract dans sa déclaration) :

```
abstract class A
{
  public abstract void f1() ;
  public abstract void f2 (char c) ;
  .....
}
// abstract obligatoire ici
abstract class B extends A
{
   // définition de f1
  public void f1 () { ..... }
  // pas de définition de f2
  .....
}
```

Ici, B définit f1, mais pas f2. La classe B reste abstraite (même si on ne l'a pas déclarée ainsi).

5. Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites. Notez que, toutes les classes dérivant de Object, nous avons utilisé implicitement cette règle dans tous les exemples précédents.

3.7.2 Intérêt des classes abstraites

Le recours aux classes abstraites facilite largement la conception orientée objet. En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

— soit sous forme d'une implémentation complète de méthodes (non abstraites) et de attributs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
— soit sous forme d'interface de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

C'est cette certitude de la présence de certaines méthodes qui permet d'exploiter le polymorphisme, et ce dès la conception de la classe abstraite, alors même qu'aucune classe dérivée n'a peut-être encore été créée. Notamment, on peut très bien écrire des canevas recourant à des méthodes abstraites. Par exemple, si vous avez défini :

```
abstract class X
{ // ici, f n'est pas encore définie
  public abstract void f();
}
```

vous pouvez écrire une méthode (d'une classe quelconque) telle que :

```
void algo (X x)
{
  .....
  // appel correct ; accepté en compilation
  x.f() ;
  // on est sûr que tout objet d'une classe dérivée de X
  .....
  // disposera bien d'une méthode f
}
```

Bien entendu, la redéfinition de f devra, comme d'habitude, respecter la sémantique prévue dans le contrat de X.

3.7.3 Exemple

Voici un exemple illustrant l'emploi d'une classe abstraite appelée *Imprimable*, dotée d'une seule méthode abstraite imprimer. Deux classes Entier et Flottant dérivent de cette classe. La méthode main utilise un tableau hétérogène d'objets de type Affichable qu'elle remplit en instanciant des objets de type Entier et Flottant.

```
abstract class Imprimable
{
  abstract public void imprimer();
}
class Entier extends Imprimable
{
  private int valeur;
  public Entier(int n)
   {
     valeur=n ;
   }
  public void imprimer()
   {
     System.out.println("Je␣suis␣un␣entier␣de␣valeur␣" + valeur) ;
   }
}
class Flottant extends Imprimable
{
  private float valeur;
  public Flottant(float x)
    {
     valeur=x;
    }
  public void imprimer()
   {
     System.out.println("Je␣suis␣un␣flottant␣de␣valeur␣" + valeur);
   }
}
```

```
public class Application
{
 public static void main (String[] args)
  {
        Imprimable[] tab;
        tab=new Imprimable[3];
        tab[0]=new Entier(25);
        tab[1]=new Flottant(1.25f);
        tab[2]=new Entier(42);
        for (int=i=0; i<3; i++)
        tab[i].imprimer();
  }
}
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

3.8 Les interfaces

Nous avons vu précédemment comment une classe abstraite permettait de définir dans une classe de base des fonctionnalités communes à toutes ses descendantes, tout en leur imposant de redéfinir certaines méthodes. Si l'on considère une classe abstraite n'implantant aucune méthode et aucun attribut (hormis des constantes), on aboutit à la notion d'interface. En effet, une interface définit les en-têtes d'un certain nombre de méthodes, ainsi que des constantes. Cependant, nous allons voir que cette dernière notion se révèle plus riche qu'un simple cas particulier de classe abstraite. En effet :

- une classe pourra implémenter plusieurs interfaces (alors qu'une classe ne pouvait dériver que d'une seule classe abstraite),
- la notion d'interface va se superposer à celle de dérivation, et non s'y substituer,
- les interfaces pourront se dériver,
- on pourra utiliser des variables de type interface.

3.8.1 Mise en oeuvre d'une interface

La définition d'une interface se présente comme celle d'une classe. On y utilise simplement le mot-clé interface à la place de *class* :

```
public interface I
{
// en tête d'une méthode f (public abstract facultatifs)
   void f(int n);
// en tête d'une méthode g (public abstract facultatifs)
   void g();
}
```

Une interface peut être dotée des mêmes droits d'accès qu'une classe (public ou droit de paquetage). Dans la définition d'une interface, on ne peut trouver que des en-têtes de méthodes (cas de f et g ici) ou des constantes (nous reviendrons plus loin sur ce point). Par essence, les méthodes d'une interface sont abstraites (puisqu'on n'en fournit pas de définition) et publiques (puisqu'elles devront être redéfinies plus tard). Néanmoins,

il n'est pas nécessaire de mentionner les mots-clés *public* et *abstract* (on peut quand même le faire).

Implémentation d'une interface

Lorsqu'on définit une classe, on peut préciser qu'elle implémente une interface donnée en utilisant le mot-clé *implements*, comme dans :

```
class A implements I
{
  // A doit (re)définir les méthodes f et g
 // prévues dans l'interface I
}
```

Ici, on indique que A doit définir les méthodes prévues dans l'interface I, c'est-à-dire f et g. Si cela n'est pas le cas, on obtiendra une erreur de compilation (attention : on ne peut pas différer cette définition de méthode, comme on pourrait éventuellement le faire dans le cas d'une classe abstraite). Une même classe peut implémenter plusieurs interfaces :

```
public interface I1
{
  void f();
}
public interface I2
{
  int h();
}
class A implements I1, I2
{
  // A doit obligatoirement définir
 // les méthodes f et h prévues dans I1 et I2
}
```

3.8.2 Interface et classe dérivée

La clause implements est une garantie qu'offre une classe d'implémenter les fonctionnalités proposées dans une interface. Elle est totalement indépendante de l'héritage ; autrement dit, une classe dérivée peut implémenter une interface (ou plusieurs) :

```
interface I
{
  void f(int n);
  void g();
}
class A { ..... }
class B extends A implements I
{
   // les mméthodes f et g doivent
   // soit être déjà définies dans A,
   // soit définies dans B
}
```

On peut même rencontrer cette situation :

```
interface I1 { ..... }
interface I2 { ..... }
class A implements I1 { ..... }
class B extends A implements I2 { ..... }
```

3.8.3 Interfaces et constantes

L'essentiel du concept d'interface réside dans les en-têtes de méthodes qui y figurent. Mais une interface peut aussi renfermer des constantes symboliques qui seront alors accessibles à toutes les classes implémentant l'interface :

```
interface I
{
  void f(int n) ;
  void g() ;
  static final int MAXI=100 ;
}
class A implements I
{
  // doit définir f et g
  // dans toutes les méthodes de A
  // , on a accès au symbole MAXI :
  // par exemple : if (i < MAXI) .....
}
```

Ces constantes sont automatiquement considérées comme si elles avaient été déclarées static et final. Il doit s'agir obligatoirement d'expressions constantes. Elles sont accessibles en dehors d'une classe implémentant l'interface. Par exemple, la constante MAXI de l'interface I se notera simplement I.MAXI.

3.8.4 Dérivation d'une interface

On peut définir une interface comme une généralisation d'une autre. On utilise là encore le mot-clé extends, ce qui conduit à parler d'héritage ou de dérivation, et ce bien qu'il ne s'agisse en fait que d'emboîter simplement des déclarations :

```
interface I1
{
  void f(int n);
  static final int MAXI=100;
}
interface I2 extends I1
{
  void g();
  static final int MINI=20;
}
```

En fait, la définition de I2 est totalement équivalente à :

```
interface I2
{
  void f(int n);
  void g();
  static final int MAXI=100;
  static final int MINI=20;
}
```

La dérivation des interfaces revient simplement à concaténer des déclarations. Il n'en va pas aussi simplement pour l'héritage de classes, où les notions d'accès deviennent fondamentales.

3.8.5 Conflits de noms

Considérons :

```
interface I1
{
  void f(int n);
  void g();
}
interface I2
{
  void f(float x);
  void g();
}
class A implements I1, I2
{
  // A doit définir deux méthodes f:
  // void f(int) et void f(float),
  // mais une seule méthode g : void g()
}
```

En ce qui concerne la méthode f, on voit que, pour implémenter I1 et I2, la classe A doit simplement la surdéfinir convenablement. En ce qui concerne la méthode g, en revanche, il semble qu'un conflit de noms apparaisse. En fait, il n'en est rien puisque les deux interfaces I1 et I2 définissent le même en-tête de méthode g ; il suffit donc que A définisse la méthode requise (même si elle elle demandée deux fois !). En revanche, considérons maintenant :

```
interface I1
{
  void f(int n);
  void g();
}
interface I2
{
  void f(float x);
  int g() ;
}
class A implements I1, I2
{
  // pour satisfaire à I1 et I2, A devrait
  // contenir à la fois une méthode
  // void g() et une méthode int g(),
  // ce qui n'est pas possible
  // d'après les règles de redéfinition
}
```

Cette fois, une classe ne peut implémenter à la fois I1 et I2.

3.8.6 Classes enveloppes

Comme on a pu le voir, les objets (instances d'une classe) et les variables (d'un type primitif) ne se comportent pas exactement de la même manière. Par exemple :

- l'affectation porte sur l'adresse d'un objet, sur la valeur d'une variable,
- les règles de compatibilité de l'affectation se fondent sur une hiérarchie d'héritage pour les objets, sur une hiérarchie de type pour les variables,
- le polymorphisme ne s'applique qu'aux objets,
- comme nous le verrons par la suite, les collections ne sont définies que pour des éléments qui sont des objets.

Les classes enveloppes (wrappers en anglais) vont permettre de manipuler les types primitifs comme des objets. Plus précisément, il existe des classes nommées Boolean, Character, Byte, Short, Integer, Long, Float et Double qui encapsulent des valeurs du type primitif correspondant (boolean, char, byte, short, int, long, float et double).

Construction et accès aux valeurs

Toutes les classes enveloppes disposent d'un constructeur recevant un argument d'un type primitif :

```
// nObj contient la référence à un objet
// de type Integer encapsulant la valeur 12
Integer nObj=new Integer (12);
// xObj contient la référence à un objet
// de type Double encapsulant la valeur 5.25
Double xObj=new Double (5.25);
```

Elles disposent toutes d'une méthode de la forme xxxValue (xxx représentant le nom du type primitif) qui permet de retrouver la valeur dans le type primitif correspondant :

```
// n contient 12
int n=nObj.intValue();
// x contient 5.25
double x=xObj.doubleValue();
```

Nous verrons un peu plus loin que ces instructions peuvent être abrégées grâce aux facilités dites de "boxing/unboxing" automatiques introduites par le JDK 5.0. Ces classes enveloppes sont finales (on ne peut pas créer de classes dérivées) et inaltérables puisque les valeurs qu'elles encapsulent ne sont pas modifiables. Les six classes à caractère numérique dérivent de la classe **Number**.

Comparaisons avec la méthode equals

On n'oubliera pas que l'opérateur == appliqué à des objets se contente d'en comparer les adresses. Ainsi, avec :

```
Integer nObj1=new Integer (5);
Integer nObj2=new Integer (5);
```

il est probable que l'expression $nObj1 == nObj2$ aura la valeur **false**. Notez que cela n'est toutefois pas certain car rien n'interdit au compilateur de n'implémenter qu'une seule fois des valeurs identiques. ; autrement dit, dans le cas présent, il peut très bien ne créer qu'un seul objet de type **Integer** contenant la valeur 5. En revanche, la méthode **equals** a bien été redéfinie dans les classes enveloppes, de manière à comparer

effectivement les valeurs correspondantes. Ici, l'expression $nObj1.equals(nObj2)$ aura toujours la valeur **true**.

3.9 Exercices

3.9.1 Définition d'une classe dérivée, droits d'accès

Nous avons à notre disposition la classe suivante :

```
class Personne
{
 private String nom, prenom;
 private int age;
 public void naissance(String nom, String prenom)
  {this.nom=nom; this.prenom=prenom;}
 public void croissance(int cr) {age+=cr;}
 public int getNom() {return nom;}
 public int getPrenom() {return prenom;}
 public int getAge() {return age;}
}
```

1. Réaliser une classe **PersonneX**, héritée de la classe **Personne** disposant d'une méthode ***informations*** affichant les informations d'une **Personne**.
2. Ecrire une classe **Application** utilisant les deux classes **Personne** et **PersonneX**.
3. Que se passerait-il si la classe Personne ne disposait pas des méthodes getNom, getPrenom et getAge ?

3.9.2 Définition d'une classe dérivée, droits d'accès(2)

Nous avons à notre disposition la classe suivante :

```
class Personne
{
 private String nom, prenom;
 private int age;
 public void SetInformations(String nom, String prenom)
  {this.nom=nom; this.prenom=prenom;}
 public void croissance (int cr) {age+=cr;}
 public void informations()
 {
  System.out.print("Je␣suis:␣" + nom + "␣" + prenom);
  System.out.println("␣age␣de:␣"+age);
 }
}
```

Réaliser une classe **Etudiant**, dérivée de **Personne** permettant de manipuler des personnes définis par leurs nom, prenom, age et une spécialité. On y prévoira les méthodes suivantes :

— **setInformations** pour définir les nom, prénom, age et la spécialité d'un objet de type Etudiant,
— **setSpecialite** pour définir la spécialité d'un étudiant,

— **informationsEtudiant** pour afficher les informations et la spécialité d'un objet de type Etudiant.

Écrire une classe Application utilisant les classes Etudiant et Personne.

3.9.3 Héritage et appels de constructeurs

Nous avons à notre disposition la classe suivante :

```
class Personne
{
private String nom, prenom;
 private int age;
 public void Personne(String nom, String prenom, int age)
  {this.nom=nom; this.prenom=prenom;this.age=age;}
 public void informations()
 {
  System.out.print("Je␣suis:␣" + nom + "␣" + prenom);
  System.out.println("␣age␣de:␣"+age);
 }
}
```

Réaliser une classe Etudiant, dérivée de Personne permettant de manipuler des personnes définis par leurs nom, prenom, age et une spécialité. On y prévoira les méthodes suivantes :

1. constructeur pour définir les informations et la spécialité d'un objet de type Etudiant,
2. informationsEtudiant pour afficher les informations et la spécialité d'un objet de type Etudiant.

Écrire une classe Application utilisant la classe Etudiant.

3.9.4 Redéfinition de méthodes

```
class Personne
{private String nom, prenom;
 private int age;
 public void Personne(String nom, String prenom)
  {this.nom=nom; this.prenom=prenom;}
 public void informations()
 {System.out.print("Je␣suis:␣" + nom + "␣" + prenom);
  System.out.println("␣age␣de:␣"+age);
 }
}
```

Réaliser une classe **Etudiant**, dérivée de **Personne** permettant de manipuler des personnes définis par leurs nom, prenom, age et une spécialité. On y prévoira les méthodes suivantes :

— constructeur pour définir les nom, prenom, age et spécialité d'un objet de type Etudiant,

— **informations** pour afficher les nom, prenom, age et spécialité d'un objet de type Etudiant.

3.9.5 Construction et initialisation d'une classe dérivée

Quels résultats fournit la classe Application ?

```
class A
{
 public int x ; // Attention, c'est juste pour demonstration
 public int y=10 ;//  que nous avons decapsule x et y,
 public A (int h)
 { System.out.println ("Debut␣A␣-␣x="+x+"␣y="+y) ;
   x=h ;
   System.out.println ("Fin␣A␣-␣x="+x +"␣y="+y) ;
 }
}
class B extends A
{
 public int z=25;
 public final int n=5;
 public B (int x, int k)
 { super(x) ;
   System.out.println ("Debut␣B␣-␣x="+x+"␣y="+y+"␣z="+z) ;
   y=k ;
   z=n+1;
   System.out.println ("Fin␣B␣-␣x="+x+"␣y="+y+"␣z="+z) ;
 }
}
public class Application
{
 public static void main (String args[])
  { A a= new A(5);
    B b= new B(5,3);
  }
}
```

3.9.6 Dérivations successives et redéfinition

Quels résultats fournit la classe Application ?

```
class A
{
 public void afficher() {System.out.println ("afficher␣de␣A");}
}
class B extends A {}
class C extends A
{
 public void afficher() {System.out.println ("afficher␣de␣C");}
}
class D extends C
{
 public void afficher() {System.out.println ("afficher␣de␣D");}
}
class E extends B {}
class F extends C {}
public class Application
{
 public static void main (String arg[])
 { A a=new A(); a.afficher();
   B b=new B(); b.afficher();
   C c=new C(); c.afficher();
   D d=new D(); d.afficher();
   E e=new E(); e.afficher();
   F f=new F(); f.afficher();
 }
}
```

3.9.7 Dérivations successives et surdéfinition

Quels résultats fournit la classe Application ?

```
class A
{
  public void f(double d) {System.out.print ("A.f(double="+d+")␣");}
}
class B extends A {}
class C extends A
{
  public void f(long l) {System.out.print ("C.f(long="+l+")␣");}
}
class D extends C
{
  public void f(int i) {System.out.print ("D.f(int="+i+")␣");}
}
class E extends B {}
class F extends C
{
  public void f(float x) {System.out.print ("F.f(float="+f+")␣");}
  public void f(int n) {System.out.print ("F.f(int="+i+")␣");}
}
public class Application
{
  public static void main (String arg[])
   {
        byte b=1; short s=2; int i=3; long l=4;
        float f=5.f; double d=6.;
        A a=new A(); a.f(b); a.f(f); System.out.println();
        B b=new B(); b.f(b); a.f(f); System.out.println();
        C c=new C(); c.f(b); c.f(l); c.f(f); System.out.println();
        D d=new D(); d.f(b); c.f(l); c.f(d); System.out.println();
        E e=new E(); e.f(b); e.f(l); e.f(d); System.out.println();
        F f=new F(); f.f(b); f.f(i); f.f(f); f.f(d); f.f(s);
   }
}
```

3.9.8 Bases du polymorphisme

Quels résultats fournit la classe Application ?

```
class A
{
 public void afficher() {System.out.print ("afficher␣de␣A");}
}
class B extends A {}
class C extends A
{
  public void afficher() {System.out.print ("afficher␣de␣C");}
}
class D extends C
{
  public void afficher() { System.out.print ("afficher␣de␣D");}
}
class E extends B {}
class F extends C {}

public class Application
{ public static void main (String arg[])
{
  A a=new A(); a.afficher(); System.out.println();
  B b=new B(); b.afficher();
  a=b; a.afficher(); System.out.println();
  C c=new C(); c.afficher();
  a=c ; a.afficher(); System.out.println();
  D d=new D(); d.afficher();
  a=d ; a.afficher();
  c=d ; c.afficher(); System.out.println();
  E e=new E(); e.afficher();
  a=e; a.afficher();
  b=e; b.afficher(); System.out.println();
  F f=new F(); f.afficher();
  a=f; a.afficher();
  c=f; c.afficher();
}
}
```

Certaines possibilités d'affectation entre objets des types classes A, B, C, D, E et F ne figurent pas dans le programme ci-dessus. Pourquoi ?

3.9.9 Polymorphisme et surdéfinition

Quels résultats fournit la classe Application ?

```
class A
{
  public void f(double x) {System.out.print ("A.f(double="+x+")");}
}
class B extends A {}
class C extends A
{
  public void f(long q) {System.out.print ("C.f(long="+q+")");}
}
class D extends C
{
  public void f(int n) {System.out.print ("D.f(int="+n+")");}
}
class F extends C
{
  public void f(float x) {System.out.print ("F.f(float="+x+")");}
  public void f(int n) {System.out.print ("F.f(int="+n+")");}
}
public class Application
{
 public static void main (String arg[])
 {
   byte bb=1 ; short p=2 ; int n=3 ; long q=4 ;
   float x=5.f ; double y=6. ;
   System.out.println ("-----A-----") ;
   A a=new A(); a.f(bb); a.f(x); System.out.println();
   System.out.println ("-----B-----");
   B b=new B(); b.f(bb); b.f(x); System.out.println();
   a=b; a.f(bb); a.f(x); System.out.println();
   System.out.println ("-----C-----");
   C c=new C(); c.f(bb); c.f(q); c.f(x); System.out.println();
   a=c; a.f(bb); a.f(q); a.f(x); System.out.println();
   System.out.println ("-----D-----");
   D d=new D(); d.f(bb); c.f(q); c.f(y); System.out.println();
   a=c; a.f(bb); a.f(q); a.f(y); System.out.println();
   System.out.println ("-----F-----");
   F f=new F(); f.f(bb); f.f(n); f.f(x); f.f(y);
   System.out.println();
   a=f; a.f(bb); a.f(n); a.f(x); a.f(y); System.out.println();
   c=f; c.f(bb); c.f(n); c.f(x); c.f(y);
 }
}
```

3.9.10 Limites du polymorphisme

Soit les classes Personne et Etudiant :

```
class Personne
{
 private String nom, prenom;
 private int age;
 public void Personne (String nom, String prenom)
  {this.nom=nom; this.prenom=prenom;}
 public static boolean identiques (Personne x, Personne y)
  { return ((x.nom==y.nom) && (x.prenom==y.prenom) && (x.age==y.age));}
 public boolean identique (Personne x)
  { return ((nom==x.nom) && (prenom==x.prenom) && (age==x.age));}
}
class Etudiant extends Personne
{
 private String  specialite ;
 public Etudiant (String nom, String prenom, String specialite)
  { super(nom, prenom); this.specialite=specialite;}
}
```

1. Quels résultats fournit la classe application ci-dessous ?

```
public class Apllication
{ public static void main (String args[])
  { Personne p = new Personne ("A", "B") ;
    Etudiant e1 = new Etudiant("A", "B", 'Info') ;
    Etudiant e2 = new Etudiant("A", "B", 'Maths') ;
    System.out.println (e1.identique(e2)) ;
    System.out.println (p.identique(e1)) ;
    System.out.println (e1.identique(p)) ;
    System.out.println (Personne.identiques(e1, e2)) ;
  }
}
```

2. Doter la classe Etudiant d'une méthode statique identiques et d'une méthode identique fournissant toutes les deux la valeur true lorsque les deux personnes concernés ont à la fois les mêmes informations et la même spécialité. Quels résultats fournira alors la classe Application précédente ?

3.9.11 Classes abstraites

On souhaite disposer d'une hiérarchie de classes permettant de manipuler des algorithmes. On veut qu'il soit toujours possible d'étendre la hiérarchie en dérivant de nouvelles classes mais on souhaite pouvoir imposer que ces dernières disposent toujours des méthodes suivantes :

— void resultat()
— void solution(String probleme)

1. Écrire la classe abstraite Algorithme qui pourra servir de classe de base à toutes ces classes.
2. Écrire une classe Factorielle utilisant la classe Algorithme.

Chapitre 4

La gestion des exceptions

Lorsque un programme Java est en cours d'exécution, il peut rencontré certaines circonstances exceptionnelles qui peuvent compromettre son exécution ; il peut recevoir par exemple des données incorrectes. Lors du conception du programme certainement le développeur essai d'examiner toutes les situations possibles, Mais il risque d'oublier certaines situations qui génère des blocages au sein du programme.

Java est doté d'un gestionnaire d'exception, un mécanisme très puissant qui permet :

— de séparer la détection d'une erreur de son traitement,

— de gérer séparément les erreurs dans le code, cela va certainement contribuer à la lisibilité du code source.

D'une façon plus générale, une exception est une interruption déclenchée par une instruction **throw**. Il y a alors un branchement à un "gestionnaire d'exception". L'aiguillage au bon gestionnaire est fait en fonction du type de l'objet mentionné à **throw** (de façon comparable au choix d'une méthode surdéfinie).

4.1 Premier exemple d'exception

4.1.1 Déclencher une exception avec throw

Soit une classe Personne, munie d'un constructeur à trois arguments et d'une méthode afficher. Nous pouvons, au sein du constructeur, vérifier la validité du paramètre age fournis. Lorsque l'age est incorrect, nous déclenchons une exception à l'aide de **throw**. À celle-ci, nous devons fournir un objet dont le type servira ultérieurement à identifier l'exception concernée. Nous créons donc une classe *ErrConst*. Java impose que cette classe soit dériver de la super classe *Exception*. Pour l'instant, nous n'y plaçons aucun attribut dans cette classe :

```
class ErrConst extends Exception
{ }
```

Pour lancer une exception de ce type au sein de notre constructeur, nous fournirons à l'instruction throw un objet de type Errconst, par la façon suivante :

```
throw new ErrConst() ;
```

En définitive, le constructeur de la classe Personne est présenté comme suite :

```
class Personne
{
  public Personne(String nom, String prenom, int age) throws ErrConst
   {
     // lance une exception de type ErrConst
     if (age<0) throw new ErrConst();
     this.nom=nom; this.prenom=prenom; this.age=age;
   }
```

Notez la présence de throws ErrConst, dans l'en-tête du constructeur, qui précise que la méthode est susceptible de déclencher une exception de type ErrConst. Cette indication est obligatoire en Java, à partir du moment où l'exception en question n'est pas traitée par la méthode elle-même. En résumé, voici la définition complète de nos classes Personne et ErrConst :

```
class Personne
{
  private String nom, prenom;
  private int age;
  public Personne(String nom, String prenom, int age) throws ErrConst
    {
      if (age<0) throw new ErrConst();
        this.nom=nom; this.prenom=prenom; this.age=age;
    }
  public void afficher()
    {
       System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
    }
}
class ErrConst extends Exception
{ }
```

4.1.2 Utilisation d'un gestionnaire d'exception

Nous avons à notre disposition la classe *Personne*, voyons maintenant comment procéder pour gérer les éventuelles exceptions de type ErrConst. Pour cela, il faut :

— Utiliser dans le bloque "try" les instructions dans lesquelles on risque de déclencher une telle exception ; un tel bloc se présente ainsi :

```
try
{
// instructions
}
```

— Suivre le bloc "try" par les différents gestionnaires d'exception (dans notre premier exemple nous aurons besoin d'un seul gestionnaire). Chaque gestionnaire est précédée d'une entête introduite par le mot-clé catch. Ci-dessous un exemple d'un gestionnaire d'exception :

```
catch (ErrConst e)
{
System.out.println ("Erreur␣construction␣");
System.exit(1);
}
```

L'exemple précédant se contente d'afficher un message et d'interrompre l'exécution du programme en appelant la méthode standard System.exit(1).

4.1.3 exemple complet

```
public class Personne
{
  private String nom, prenom;
  private int age;
  public Personne(String nom, String prenom, int age) throws ErrConst
    {
      if (age<0) throw new ErrConst();
        this.nom=nom; this.prenom=prenom; this.age=age;
    }
  public void afficher()
    {
       System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
    }
}
public class ErrConst extends Exception {}
public class Application
{
public static void main (String args[])
   { try
      {
        Personne p=new Personne("Toumi", "Maria", 5);
        p.afficher();
        p=new Personne("Toumi", "Maria", -5);
        p.imprimer() ;
      }
     catch (ErrConst e)
     { System.out.println ("Erreur␣construction␣");
      System.exit (1) ;
     }
   }
}
```

```
Je suis Toumi Maria age: 5 ans
Erreur construction
```

4.1.4 Propriétés du gestionnaire d'exception

L'exemple précédant était très restrictif pour différentes raisons :

- Nous avons déclenché et traité qu'un seul type exception ;
- le gestionnaire d'exception ne recevait aucune information ;
- nous utilisons pas les méthodes et les attributs de la classe *Exception* dont dérive notre classe *ErrConst* ;
- le gestionnaire d'exception se contentait d'interrompre.

Nous pouvons noter que le gestionnaire d'exception est défini indépendamment des méthodes susceptibles de la déclencher. Ainsi, à partir du moment où la définition d'une classe est séparée de son utilisation (ce qui est souvent le cas en pratique), il est tout à fait possible de prévoir un gestionnaire différent d'une utilisation à une autre de la même classe. Dans l'exemple précédent, tel utilisateur peut vouloir afficher un

message avant de s'interrompre, tel autre préférera ne rien afficher ou encore tenter de trouver une solution par défaut... D'autre part, le bloc *try* et les gestionnaires associés doivent être contigus. Cette construction est erronée :

```
try
{
  .....
}
 // erreur : catch doit être contigu au bloc try
catch (ErrConst)
{ ..... }
```

Maintenant, dans notre exemple, le bloc *try* couvre toute la méthode main. Ce n'est nullement une obligation et il est même théoriquement possible de placer plusieurs blocs try dans une même méthode :

```
void principe()
{
   .....
  try {
       ..... // ici, les exceptions ErrConst sont traitées
      }
      catch (ErrConst)
      { ..... }
    ..... // ici, elles ne le sont plus
   try { .....
       }
       catch (ErrConst)
       { ..... } // ici, elles le sont de nouveau
       ..... // ici, elles ne le sont plus
}
```

4.2 Gestion de plusieurs exceptions

Nous proposons un exemple plus complet dans lequel on peut déclencher et traiter deux types d'exceptions. Pour ce faire, nous considérons une classe Personne munie :

- du constructeur précédent, déclenchant toujours une exception ErrConst,
- d'une méthode grandir qui s'assure que le agrandissement ne conduit pas à une valeur négative de l'attribut *age* ; s'il est le cas, elle déclenche une exception ErrAge :

```
public void grandir (int g) throws ErrAge
{
  if ((age+g)<0) throw new ErrAge();
  age+=g;
}
```

Lors de l'utilisation de notre classe Personne, nous pouvons détecter les deux potentielles exceptions, ErrConst et ErrAge, en procédant ainsi :

```
try
{
// bloc dans lequel on souhaite détecter
// les exceptions ErrConst et ErrAge
}
// gestionnaire de l'exception ErrConst
catch (ErrConst e)
{
 System.out.println ("Erreur construction ");
 System.exit(1) ;
}
// gestionnaire de l'exception ErrAge
catch (ErrAge e)
{
 System.out.println ("Erreur d'agrandissement");
 System.exit(1) ;
}
```

Voici un exemple complet dans lequel nous provoquons volontairement une exception ErrAge ainsi qu'une exception ErrConst :

```
public class Personne{
  private String nom, prenom;
  private int age;
  public Personne(String nom, String prenom, int age) throws ErrConst
    {
     if (age<0) throw new ErrConst();
     this.nom=nom; this.prenom=prenom; this.age=age;
    }
    public void grandir(int g) throws ErrAge
    {
     if ((age+g)<0) throw new ErrAge() ;
     age+=g;
    }
  public void afficher()
    {
       System.out.println ("Je suis "+nom +" "+ prenom+" age: "+age+" ans");
    }
}
class ErrConst extends Exception {}
class ErrAge extends Exception {}
public class Application
{
  public static void main (String args[])
   {
   try
     {
        Personne p=new Personne("Toumi", "Maria", 5);
        p.afficher();
        p.grandir(-20);
        p=new Personne("Toumi", "Maria", -5);
        p.imprimer();
     }
     catch (ErrConst e)
     {
```

```
        System.out.println ("Erreur construction");
        System.exit(1) ;
      }
      catch (ErrAge e)
      { System.out.println ("Erreur agrandissement");
        System.exit(1) ;
      }
    }
}
```

```
Je suis en Toumi Maria age: 5 ans
Erreur agrandissement
```

Bien entendu, comme la première exception (ErrAge) provoque la sortie du bloc try (et, de surcroît, l'arrêt de l'exécution), nous n'avons aucune chance de mettre en évidence celle qu'aurait provoquée la tentative de construction d'une Personne par l'appel new Personne("Toumi", "Maria", -5).

4.3 Transmission d'information au gestionnaire d'exception

Nous pouvons transmettre une information au gestionnaire d'exception :
— par le biais de l'objet fourni dans l'instruction **throw**,
— par l'intermédiaire du constructeur de l'objet exception.

4.3.1 Par objet fourni à throw

Nous avons vu précédemment que l'objet fourni à l'instruction **throw** sert à choisir le bon gestionnaire d'exception. Mais cet objet est aussi récupéré par le gestionnaire d'exception sous la forme d'un argument utilisé pour passer l'information. Il suffit pour cela de prévoir les attributs ou les méthodes correspondantes . Ci-dessous une adaptation de l'exemple précédent, dans lequel nous dotons la classe ErrConst d'attribut age permettant de transmettre l'age reçu par le constructeur de Personne. Sa valeur est fixé lors de la construction d'un objet de type ErrConst et elle est récupérée directement par le gestionnaire d'exception.

```
public class Personne{
  private String nom, prenom;
  private int age;
  public Personne(String nom, String prenom, int age) throws ErrConst
    {
     if (age<0) throw new ErrConst(age);
     this.nom=nom; this.prenom=prenom; this.age=age;
    }
  public void afficher()
    {
       System.out.println ("Je␣suis␣"+nom +"␣"+ prenom+"␣age:␣"+age+"␣ans");
    }
}
class ErrConst extends Exception
{
  public int age;
   ErrConst (int age)
   {
    this.age=age;
   }
}
public class Application
{
  public static void main (String args[])
   {
   try
     {
        Personne p=new Personne("Toumi", "Maria", 5);
        p.afficher();
        p=new Personne("Toumi", "Maria", -5);
        p.imprimer();
     }
     catch (ErrConst e)
     {
       System.out.println ("Erreur␣construction␣Personne");
           System.out.println ("␣age␣souhaite␣impossible");
           System.exit(1);
     }
    }
}
```

```
Je suis en Toumi Maria age: 5 ans
Erreur construction Personne
age souhaite impossible
```

4.3.2 Par constructeur de la classe exception

Parfois nous pouvons se contenter de transmettre un "message" au gestionnaire, sous forme de chaîne de caractère. Nous pouvons utiliser la classe Exception, celle-ci dispose d'un constructeur à un argument de type String dont on peut récupérer la valeur à l'aide de la méthode getMessage. Ci-dessous une adaptation dans ce sens de l'exemple précédent :

```
public class Personne{
  private String nom, prenom;
  private int age;
  public Personne(String nom, String prenom, int age) throws ErrConst
    {
     if (age<0) throw new ErrConst("Erreur construction") ;
     this.nom=nom; this.prenom=prenom; this.age=age;
    }
  public void afficher()
    {
       System.out.println ("Je suis "+nom +" "+ prenom+" age: "+age+" ans");
    }
}
class ErrConst extends Exception
{
   ErrConst (String message)
   {
    super(message) ;
   }
}
public class Application
{
  public static void main (String args[])
   {
   try
     {
        Personne p=new Personne("Toumi", "Maria", 5);
        p.afficher();
        p=new Personne("Toumi", "Maria", -5);
        p.imprimer();
     }
     catch (ErrConst e)
     {
       System.out.println(e.getMessage());
           System.exit(1);
     }
    }
}
```

```
Je suis en Toumi Maria age: 5 ans
Erreur construction
```

4.4 Bloc finally

Java permet d'introduire, à la suite d'un bloc **try**, un bloc particulier d'instructions qui seront toujours exécutées :

- soit après la fin "naturelle" du bloc **try**, si aucune exception n'a été déclenchée (il peut s'agir d'une instruction de branchement inconditionnel telle que break ou continue),
- soit après le gestionnaire d'exception (à condition, bien sûr, que ce dernier n'ait pas provoqué d'arrêt de l'exécution).

Ce bloc est introduit par le mot-clé **finally** et doit obligatoirement être placé après le dernier gestionnaire. Bien entendu, cette possibilité n'a aucun intérêt lorsque les exceptions sont traitées localement. Considérez par exemple :

```
try { ..... }
catch (Ex e) { ..... }
finally
{
// instructions A
}
```

Ici, le même résultat pourrait être obtenu en supprimant tout simplement le mot-clé finally et en conservant les instructions A (avec ou sans bloc) à la suite du gestionnaire. En revanche, il n'en va plus de même dans :

```
void f (...) throws Ex
{ .....
try
{ ..... }
finally
{ // instructions A }
.....
}
```

Ici, si une exception Ex se produit dans f, on exécutera les instructions A du bloc **finally** avant de se brancher au gestionnaire approprié. Sans la présence de **finally**, ces mêmes instructions ne seraient exécutées qu'en l'absence d'exception dans le bloc **try**. D'une manière générale, le bloc **finally** peut s'avérer précieux dans le cadre de ce que l'on nomme souvent l'acquisition de ressources. On range sous ce terme toute action qui nécessite une action contraire pour la bonne poursuite des opérations : la création d'un objet, l'ouverture d'un fichier, le verrouillage d'un fichier partagé... Toute ressource acquise dans un programme doit pouvoir être convenablement libérée, même en cas d'exception. Le bloc **finally** permet de traiter le problème puisqu'il suffit d'y placer les instructions de libération de toute ressource allouée dans le bloc **try**.

4.5 Exercices

4.5.1 Traitement d'exception

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

1. d'un constructeur à un argument de type int qui générera une exception de type ErreurEntier (une classe à définir) lorsque la valeur reçue ne conviendra pas,
2. d'une méthode getN fournissant sous forme d'un entier, la valeur encapsulée dans un objet de type EntierNaturel.

Écrire une classe Application qui traite l'exception ErreurEntier en affichant un message.

4.5.2 Gestion d'exception

Adapter la classe EntierNaturel et la classe Application de l'exercice précédant de manière à disposer dans le gestionnaire d'exception du type ErreurEntier de la valeur fournie à tort au constructeur.

4.5.3 Gestion de plusieurs exceptions

Que produit la classe Route lorsqu'on lui fournit en donnée :
- la valeur 0,
- la valeur 1,
- la valeur 2.

```
class ExceptionJava extends Exception
 { public int n ;
   public ExceptionJava (int n) {this.n=n;}
 }
public class Routes
{ int n;
  public static void main (String args[])
  { System.out.print ("donnez un entier:"); n=Lecture.entier();
    try
    { System.out.println ("debut premier bloc try");
      if (n!=0) throw new ExceptionJava(n);
      System.out.println ("fin premier bloc try");
    }
    catch(ExceptionJava 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 ExceptionJava(n);
     System.out.println ("fin second bloc try");
    }
    catch(ExceptionJava e)
    { System.out.println ("catch 2 - n = " + e.n);}
     System.out.println ("fin programme");
  }
}
```

4.5.4 Héritage et exceptions

Que fournit la classe Application ?

```
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 Application
{
  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");
   }
}
```