Programmation Orientée Objet

Et langage JAVA

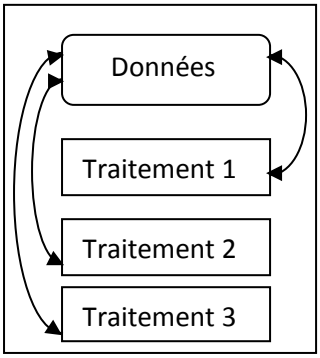
1. Introduction

Le programmeur par objets se pose en premier la question :

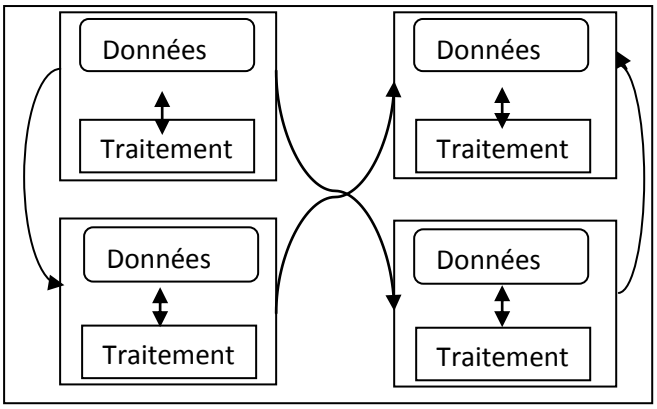
- ✓ quels sont les objets fondamentaux de l'application ?
- ✓ Quelles sont les choses et les concepts qui constituent le domaine de l'application ?

Il décrit alors chaque type d'objet sous forme de variables qui caractérisent son état puis il s'intéresse aux opérations que peuvent subir ces objets et les décrit sous forme de procédures. Ceci conduit à une **modularité** basée sur les types de données. Les traitements sont toujours définis à propos de la définition d'un type de données, et non pas isolément comme c'est le cas pour une modularité basée en priorité sur la décomposition des traitements.

Langages procéduraux



Langages orientés objet



En résumé, si la programmation structurée s'intéresse aux traitements puis aux données, la conception objet s'intéresse d'abord aux données, auxquelles elle associe ensuite les traitements. L'expérience a montré que les données sont ce qu'il y a de plus stable dans la vie d'un programme, il est donc intéressant d'architecturer le programme autour de ces données. Le concept objet permet d'obtenir des logiciels fiables, évolutifs et faciles à maintenir.

2. Concepts de base de la programmation orientée objet

2.1 Classes et Objets

Classe :

Une classe est la généralisation de la notion de type défini par l'utilisateur (les structures les unions et les énumérations), dans laquelle se trouvent associées à la fois des ***données*** et des ***méthodes***. Une classe regroupe donc un ensemble de données (qui peuvent être des variables primitives ou des objets) et un ensemble de méthodes de traitement de ces données et/ou de données extérieures à la classe.

Les données se présentent sous forme de champs désignés par des identificateurs et dotés d'un type. Ces champs sont généralement des variables qui représentent l'état de l'objet.

Les procédures, également appelées ***méthodes***, définissent les opérations possibles sur un tel objet. Ces données et procédures sont qualifiées de ***membres*** ou ***composants*** de la classe.

Objet :

Comme une *classe* (ou une structure) n'est qu'un simple type, les *objets* possèdent les mêmes caractéristiques que les variables ordinaires.

Un *objet* est donc une variable (presque) comme les autres. Il faut notamment qu'il soit déclaré avec son type. Le type d'un objet est un type complexe (par opposition aux types primitifs entier, caractère, . . .) qu'on appelle une *classe*.

Une *classe* est la définition d'un type, alors que l'*objet* est une déclaration de variable.

Après avoir créé une classe, on peut créer autant d'objets que l'on veut basés sur cette classe.

2.2 Etapes d'une conception orientée objet

L'approche objet offre une manière claire pour concevoir une architecture de modules autonomes pour une implémentation multiplateforme. Elle se déroule en trois étapes:

L'analyse:

Cette étape consiste à étudier les attentes des utilisateurs pour identifier les objets du monde réel qu'ils invoquent dans leur activité.

Exemple : Si on veut écrire un logiciel de « Gestion d'une bibliothèque », on se pose la question : de quoi est composé ce problème ?

On identifie donc à cette étape tous les objets qui interviennent dans une bibliothèque à savoir:

- Des livres
- Des journaux
- Une directrice ou un directeur
- Un bibliothécaire
- Des lecteurs

} Les objets du problème (ensembles de données)

L'étape suivante est de regrouper les objets similaires dans une même **classe :**

- ✓ même ***structure de données*** et ***méthodes de traitement.***
- ✓ valeurs différentes pour chaque objet.

Nous constatons que ces données peuvent être regroupées en 4 classes :

- Classe **livre** identifiée par : titre du livre et le nom de l'auteur;
- Classe **journal** identifiée par : titre du journal;
- Classe **employé** identifiée par : nom, prénom de l'employé et statut de l'employé ;
- Classe **lecteur** identifiée par : nom et prénom du lecteur.

La conception:

Cette étape consiste à concevoir l'organisation des classes et le choix des structures de données pour le stockage.

L'implémentation:

C'est l'étape de la réalisation pratique du logiciel, l'écriture des programmes, le débogage, les tests et la validation.

2.3 Création d'une classe

Une classe peut comporter :

- ✓ Un ensemble de déclarations d'identificateurs : ***les attributs***, définissant une structure,
- ✓ un ensemble de définitions de fonctions : ***les méthodes***, définissant un comportement.
- ✓ des ***constructeurs***, qui permettent de créer des objets ;

2.3.1 Syntaxe

La déclaration d'une classe se fait de la façon suivante :

```
class NomClasse
 {
 corps de la classe
 }
```

Par convention le nom de la classe doit débuter par une **majuscule**.

2.3.2 Attributs :

Les attributs représentent la description des données propres à chaque classe d'objets. Ceux-ci peuvent être des objets d'autres classes ou des références sur d'autres objets.

2.3.3 Méthodes :

Les méthodes représentent l'ensemble des actions, procédures, fonctions ou opérations que l'on peut associer à une classe.

Écriture des méthodes :

La ***signature*** d'une méthode contient notamment :

- un nom de méthode;
- un type de données de retour;
- des types de données pour ses arguments;
- …

Par convention le nom des méthodes et attributs commencent toujours par une **minuscule**.

Une méthode est composée de sa signature et d'un traitement associé.

```
<type de retour> <nom de méthode> (<arguments>)
        {
                <traitement associé>
        }
```

2.3.4 Encapsulation :

On parle d'*encapsulation* pour désigner le regroupement de données dans une classe. L'encapsulation de données dans un objet permet de cacher ou non leur existence aux autres objets du programme. Une donnée peut être déclarée en accès :

- **public** : les autres objets peuvent accéder à la valeur de cette donnée ainsi que la modifier ;
- **private** : les autres objets n'ont pas le droit d'accéder directement à la valeur de cette donnée (ni de la modifier). En revanche, ils peuvent le faire indirectement par des méthodes de l'objet concerné (si celles-ci existent en accès public).

2.3.5 Surcharge des méthodes

Dans Java, il est possible de créer dans une classe plusieurs méthodes qui portent le même nom mais avec différents paramètres et/ou valeurs de retour.
Cela est connu sous le nom de *surcharge des méthodes*. Java décide de la méthode à appeler en regardant la valeur de retour et les paramètres.

2.4 Création d'un objet

2.4.1 Constructeurs

Un **constructeur** est une méthode particulière invoquée lors de la création d'un objet. Cette méthode, qui peut être vide, doit donner une valeur initiale à tous les champs de son type objet.
Chaque classe doit définir un ou plusieurs **constructeurs**.
Chaque constructeur doit avoir le **même nom** que la classe où il est défini et n'a aucune valeur de retour (c'est l'objet créé qui est renvoyé).

Exemple :

```
class Date {
  private int jour, mois, an;      // 3 attributs
  final int max_mois = 12;       // un attribut constant

  public Date() {                  // définition du constructeur
    jour = mois = an = 0;       // par défaut
  }

  public Date(int j, int m, int a) {      // constructeur surchargé
    jour = j; mois = m; an = a;
  }

  public void printDate() {
    // ...
  }

  // ...
} // end class date
```

2.4.2 Instanciation

L'instanciation est l'opération qui consiste à créer un objet à partir d'une classe. Une définition de classe constitue un modèle d'objet dont on peut ensuite créer autant d'exemplaires que l'on veut (on dit également ***instances***).

En Java, la ***déclaration*** et la ***création*** des objets sont deux choses séparées. La déclaration consiste à définir un identificateur et lui associer un type afin que cet identificateur puisse désigner un objet de ce type.
En Java, un identificateur de variable de type classe est associé à une ***référence*** qui désigne un objet. Une référence est ***l'adresse*** d'un objet, mais contrairement à un pointeur (tel qu'on en trouve en C, C++ ou Pascal) la valeur d'une référence n'est ni accessible, ni manipulable : elle ne peut que permettre l'accès à l'objet qu'elle désigne.

La déclaration seule ne crée pas d'objet. Elle associe l'identificateur à une référence appelée **null** qui ne fait référence à rien.

Pour créer un objet de classe ***C***, il faut exécuter **new** qui provoque une instanciation en faisant appel à un **constructeur** de la classe instanciée.

Exemple : Date d1=new Date() ; // constructeur sans paramètres

Date d2=new Date(12,4,2014) ; // constructeur avec paramètres

Remarque importante **:** en Java, la notion de pointeur est transparente pour le programmeur. Il faut néanmoins savoir que ***toute variable désignant un objet est un pointeur***. Il s'ensuit alors que le passage d'objets comme paramètres d'une méthode est **toujours** un passage par référence. À l'inverse, le passage de variables primitives comme paramètres est toujours un passage par valeur.

2.4.3 Garbage collector « ramasse miettes »

La libération de l'espace mémoire créée dynamiquement est automatique en java. Dès que la donnée n'est plus référencée par un pointeur. C'est le mécanisme du garbage collector, aussi appelé « ramasse miettes », qui s'en charge.
Autrement dit, la destruction des objets est prise en charge par le "garbage collector". Ce dernier détruit les objets qui n'ont plus de référence

2.5 Invocation de méthodes :

En Java, une méthode ne peut pas être invoquée seule, elle est toujours appelée sur un objet.

Un point **.** sépare le nom de la méthode de l'objet sur lequel elle est invoquée.

(La syntaxe pour accéder aux attributs d'un objet est la même).

Pour qu'un tel appel soit possible, il faut que trois conditions soient remplies :

1. La variable ou la méthode appelée existe.
2. Une variable désignant l'objet visé existe et soit instanciée.
3. L'objet, au sein duquel est fait cet appel, ait le droit d'accéder à la méthode ou à la variable

Exemple :

```
…
Date d1 ;  // pas d'instanciation
Date  d2=new Date(12,4,2011);  // variable instanciée
d2.printDate();  // appel à la méthode printDate() (méthode public)
d2.jour=14;  // modification de l'attribut jour s'il n'est pas déclaré en private
…
```

Rappel : d1 et d2 contiennent des références sur l'objet créé donc il est possible d'écrire **d1=null** ; // d1 ne pointe sur aucun objet et vous constaterez que le mot clé null en JAVA est en minuscule contrairement au langage C.

2.6 Le mot-clé this :

Ce mot-clé est une référence sur l'objet en cours, c'est à dire la classe dans laquelle on se trouve

Il désigne, en cours d'exécution d'une méthode, l'objet sur lequel elle est appelée.

Aussi, l'appel à ***this*** peut être utile pour bien différencier les variables de classe des variables de méthodes.

On peut aussi accéder aux constructeurs de la classe elle-même

Exemple :

```
public class Maclasse {
  private Date date;

 public Maclasse() {
    date=new Date() ;  // appel au constructeur par défaut
}
```

```
void traitement1(Date date) {

    this() ;                // appel au constructeur de la classe

    this.date = date;  /* this.date désigne l'attribut date de cet objet
                          et date désigne le paramètre ici this permet
                          de lever l'ambiguité  */
  }
  Date traitement2( ) {
    // ...
    traitement1( this );    // passage en paramètre de l'objet courant

    // ...
    return this;        // retourne l'objet courant
  }
  // ...
} // fin Maclasse
```

2.7 Les accesseurs (set et get) :

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées « ***private*** » à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe.
Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés ***« échanges de message ».***
Un accesseur est une méthode public qui donne l'accès à une variable d'instance privé.
Pour une variable d'instance :

- il peut ne pas y avoir d'accesseur,
- un seul accesseur en lecture,
- un accesseur en lecture et un accesseur en écriture.

Remarque : Par convention, les accesseurs en lecture commencent par « get » et les accesseurs en écriture commencent par « set ».

Exemple :

```
import java.lang.Math;  // importer la classe Math
public class Point {
private double x,y;
        public Point () {
                x=y=0;
        }
        public Point (double x, double y) {
                this.x=x;this.y=y;
        }
        public double getx(){
                return x;
        }
        public double gety() {
                return y;
        }
        public void setx(double x){
                this.x=x;
        }
        public void sety(double y) {
                this.y=y ;
        }
        public void afficher() {
        System.out.println("x= " + x + " y= " + y);
        }
}
```

```
import java.util.Scanner;   /* importer la classe Scanner contenant les méthodes
                               de lecture des données */
public class Ppoint {       // classe exécutable contenant la méthode main

        public static void main(String[] args) {
           Scanner Entree=new Scanner(System.in); /* créer un objet de type
                        Scanner pour pouvoir invoquer les méthodes de lecture */

           System.out.println("Donnez les coordonnées d'un  points");
                double ax=Entree.nextDouble();
                double ay=Entree.nextDouble();
                Point p1=new Point(ax,ay); // instanciation de l'objet p1

           p1.afficher();          p1.setx(2);             p1.afficher();
        }
}
```

2.7 Composants non liés à des objets

On a parfois besoin de méthodes non associées à un objet. Une telle méthode doit être rédigée dans une classe, car il n'y a pas de méthodes isolées en Java. Cela est souvent naturel, car la classe joue alors un rôle structurant en regroupant les méthodes autour d'un même thème. Pour cela, il faut déclarer la méthode avec l'attribut **static**.

static : indique que la variable est en fait globale à toutes les instances de la classe ou des ses sous-classes[1]. Elle est directement accessible par la classe.

final : indique que la valeur de la variable ne peut pas être changée (représente en fait une constante).
Les constantes sont généralement déclarées comme ***public static final***.

Exemples:

- ***final***

```
...
public static final double PI = 3.14159265358979323846 ;
...
```

- ***Méthodes non static***

```
import java.lang.Math;
public class Point {
private double x,y;
// …
public  Point milieu( Point p){
        return new Point((x+p.x)/2,(y+p.y)/2);
}
public  double distance (Point p) {
        return Math.sqrt(Math.pow((x-p.x),2.0)+Math.pow(y-p.y),2.0));
}
public void deplace (double dx,double dy){
        setx(x+dx);
        sety(y+dy);  }
} // Fin Point

import java.util.Scanner;
public class Ppoint {
        public static void main(String[] args) {
           Scanner Entree=new Scanner(System.in);
           System.out.println("Donnez 2 points");
                Point p1=new Point(Entree.nextDouble(), Entree.nextDouble());
                Point p2=new Point(Entree.nextDouble(), Entree.nextDouble());
```

[1] Sous-classe : sera définie dans le chapitre Héritage.

```
        Point m=p1.milieu(p2);   // nom de l'objet . nom de la méthode
        m.afficher();
        System.out.println("distance "+ p1.distance(p2));
    }
}// Fin Ppoint
```

- ***Méthodes static***

```
import java.lang.Math;
public class Point {
private double x,y;

//  …

public  static Point milieu( Point p1,Point p2){
    return new Point((p1.x+p2.x)/2,(p1.y+p2.y)/2);
}
public static double distance (Point p1,Point p2) {
    return Math.sqrt(Math.pow((p2.x-p1.x),2.0)+Math.pow((p2.y-p1.y),2.0));
}
public void deplace (double dx,double dy){
    setx(x+dx);
    sety(y+dy);   }
}// Fin Point

import java.util.Scanner;
public class Ppoint {   // classe exécutable

    public static void main(String[] args) {
        Scanner Entree=new Scanner(System.in);
        System.out.println("Donnez 2 points");
            double ax=Entree.nextDouble();
            double ay=Entree.nextDouble();
            double bx=Entree.nextDouble();
            double by=Entree.nextDouble();
        //-------------------------------------------------------------------------
        Point p1=new Point(ax,ay);
        Point p2=new Point(bx,by);
        Point m=new Point(0,0);

        m=Point.milieu(p1,p2);  // nom de la classe . nom de la méthode
        m.afficher(); // nom de l'objet . nom de la méthode
        p1.deplace(1,1);
        p1.afficher();
    } // fin main
} fin Ppoint
```

- ***Attribut static***

```
import java.util.Scanner;
public class PointTab {
        public static int k;      //  Une variable static
        private int n;
        private Point[] tab;    //  Un tableau de points
        public static final int max=100;      //  Une constante static

        public PointTab(){
                tab=new Point[max];
                n=0;
        }

        // ...

}// fin PointTab
```

2.8 Package

Un grand nombre de classes, fournies par *SUN*, implémentent des données et des traitements génériques utilisables par un grand nombre d'applications. Ces classes forment l'API (*Application Programmer Interface)* du langage Java.
Toutes ces classes sont organisées en ***packages*** (ou bibliothèques) dédiés à un thème précis.

2.8.1 Les packages prédéfinies

Les classes prédéfinies en Java sont standardisées et placées dans des bibliothèques dont les noms sont eux-mêmes standard dont voici quelques unes:

- **java.lang** contient les classes de base (chaînes de caractères, mathématiques, tâches, exceptions ...)

- **java.util** contient des classes comme vecteurs, piles, files, tables ...

- **java.io** contient les classes liées aux entrées/sorties : texte et fichier

- **java.awt** contient les classes pour les interfaces (fenêtres, boutons, menus, graphique, événements ...)

- **javax.swing** contient les classes pour les interfaces (évolution de awt)

- **java.net** contient les classes relatives à internet (sockets, URL ...)

- **java.applet** contient les classes permettant de créer des applications contenues dans des pages en HTML

Pour pouvoir utiliser les classes de ces bibliothèques il faut y faire référence en début de fichier en utilisant la directive **import** suivie du nom de la classe de la bibliothèque ou de ***** pour accéder à toutes les classes de cette bibliothèque: par exemple **import java.io.***

Certaines de ces bibliothèques contiennent des sous-bibliothèques qu'il faut explicitement nommer (par exemple **java.awt.events.*** pour les événements)

Remarque : les classes de la bibliothèque de base ***java.lang*** n'ont pas besoin de la commande import.

Pour accéder à une classe d'un package donné, il faut préalablement importer cette classe ou son package. Par exemple, la classe Date appartenant au package java.util qui implémente un ensemble de méthodes de traitement sur une date peut être importée de deux manières :

• une seule classe du package est importée : import java.util.Date;
• toutes les classes du package sont importée (même les classes non utilisées) : import java.util.*;

Le programme suivant utilise la classe prédéfinie Date pour afficher la date actuelle :

```
import java.util.Date;
public class DateMain {
    public static void main(String[] args) {
        Date today = new Date();
        System.out.println("Nous sommes le " + today.toString());
    }
}
```

2.8.2 La création d'un package

Pour faciliter la réutilisation du code, Java permet de regrouper plusieurs définitions de classes dans un groupe logique appelé *package (paquet)*. Si, par exemple, vous créez un groupe de règles de gestion qui modélisent les traitements de gestion de votre entreprise, vous pouvez les réunir dans un package. Cela facilite la réutilisation du code précédemment créé.

Pour réaliser un package, on écrit un nombre quelconques de classes dans plusieurs fichiers sources d'un **même répertoire** et au début de chaque fichier on met la directive ci-dessous ou nom-de-package doit être composé des répertoires séparés par un caractère point :

package *nom-de-package*

De façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.
Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

Exemple :

Pour assigner la classe précédente à un package, nommé test.dateToday, il faut modifier le fichier de cette classe comme suit :

```
package test.dateToday;
import java.util.Date;
public class DateMain {
...
}
```

Enfin, il faut que le chemin d'accès du fichier DateMain.java corresponde au nom de son package. Celui-ci doit donc être situé dans un répertoire test/dateToday/DateMain.java accessible à partir des chemins d'accès définis lors de la compilation ou de l'exécution.

Au sein d'un package, on a accès aux classes de ce package et aux classes déclarées **public** des autres package, mais dans ce dernier cas il faut utiliser un nom absolu : ***nom-de-package.nom-de-classe***

Il est possible de désigner les classes par leur nom court, sans préciser le package, à condition d'utiliser la directive **import** :

import *nom-de-package.nom-de-classe*

ou **import *nom-de-package.**** pour désigner toutes les classes du package.

Ainsi, n'importe quel autre programme peut, pour accéder aux classes déclarées dans un même package, utiliser l'instruction : ***import nom-de-package.*;***

3. Héritage :

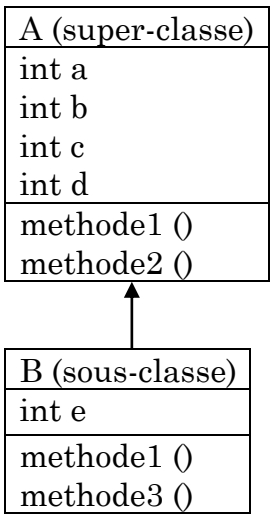
L'héritage est un mécanisme destiné à exprimer les similitudes entre classes. Il met en œuvre les principes de ***généralisation et de spécialisation*** en partageant explicitement les attributs et méthodes communs au moyen d'une hiérarchie de classes. Quand une classe hérite d'une autre classe, la classe enfant hérite automatiquement de toutes les caractéristiques (variables membre) et du comportement (méthodes) de la classe parent.

Un des avantages de la programmation objet est de pouvoir réutiliser et enrichir des classes sans avoir à tout redéfinir. La super classe contient les attributs et méthodes communs aux classes dérivées. La classe dérivée ajoute des attributs et des méthodes qui spécialisent la super classe. Si une méthode de la super classe ne convient pas, on peut la redéfinir dans la sous classe. Cette méthode redéfinie peut appeler la méthode de la super classe si elle convient pour effectuer une partie du traitement. On peut utiliser sans les définir les méthodes de la super classe si elles conviennent.

On distingue deux types d'héritage: l'héritage simple et l'héritage multiple.

3.1 Héritage simple

On a parfois besoin de définir un type d'objet similaire à un type existant, avec quelques propriétés supplémentaires. L'héritage permet de définir ce nouveau type sans tout reprogrammer : il suffit de déclarer que le nouveau type hérite du précédent et on se borne à rédiger ses fonctionnalités supplémentaires. Ceci constitue l'utilisation la plus simple de l'héritage.
Une classe fille n'a qu'une seule classe mère et elle hérite d'une partie des attributs et méthodes de celle-ci en ayant ses spécifications propres. Cette notion est celle qui est utilisée dans les méthodes objets hiérarchiques.



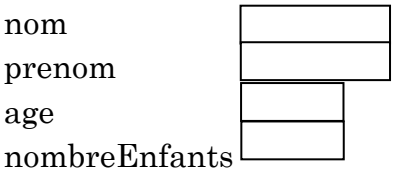
Le principe de l'héritage : la classe B hérite de la classe A.

Dans Java, l'héritage est géré avec le mot clé ***extends***. Quand une classe hérite d'une autre classe, la classe enfant étend la classe parent.

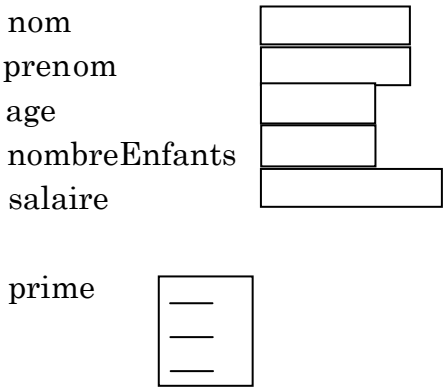
Par exemple, on peut dériver un type ***Salarié*** à partir d'un type plus général ***Personne :***

- La notation ***class Salarié extends Personne {...}*** indique que la classe **Salarié** ***hérite*** de la classe **Personne**.
- On dit également que **Salarié** est une classe ***dérivée*** de **Personne**.
- **Personne** est une ***classe de base*** de **Salarié**.
- Le type **Salarié** possède toutes les propriétés du type **Personne**, mêmes composants (données et méthodes), plus quelques nouvelles, le champ **salaire** et la fonction **prime()**.

Une personne

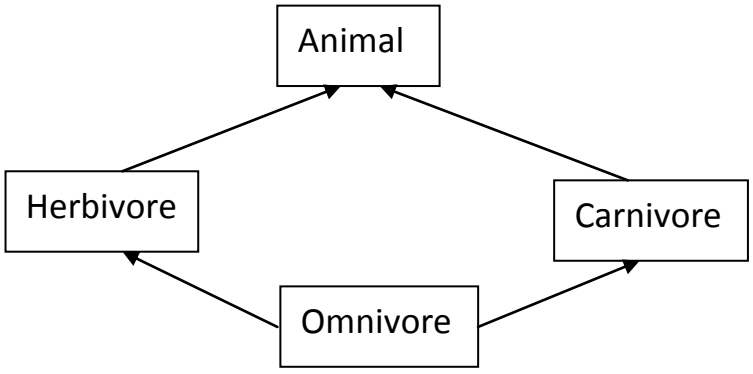


Un salarié



3.2 Héritage multiple

Une classe fille hérite de plusieurs classes mères. Elle hérite d'une partie des attributs et des méthodes de chacune de ses classes mères, en plus de ses spécifications propres. Beaucoup de langages à objets ne proposent pas d'héritage multiple. **Ce n'est pas le cas de Java** : une classe ne peut hériter que d'une seule classe.



Résumé :

L'héritage permet de spécialiser une classe sans avoir à tout récrire, en utilisant, sans modification, les méthodes de la classe mère si elles conviennent ou en récrivant les méthodes qui ne conviennent plus, sachant qu'une partie du traitement peut souvent être faite par les méthodes de la super classe.

- L'héritage permet de créer de nouvelles classes à partir de classes existantes.
- Les classes dérivées obtenues sont des « classes filles », « classes dérivées » ou « sous classes ».
- Les classes d'origine sont dites des « super classes », « classes de base » ou « classes parentes ».
- Les sous classes peuvent hériter des attributs de la super classe dont elles sont issues.
- Les sous classes peuvent avoir des attributs et des méthodes qui leur sont propres.
- Certaines méthodes d'une super-classe peuvent être redéfinies dans les sous classes issues de celle-ci.

3.3 Utilisation de *this* et de *super*

Si vous avez fréquemment besoin d'appeler explicitement le constructeur de la classe supérieure, utilisez le mot clé Java ***Super()***. Il appelle le constructeur de la classe parent, doté des paramètres nécessaires.

Si vous créez plusieurs constructeurs, vous ne voulez pas dupliquer le code commun. Pour cela, appelez le constructeur d'une même classe avec le mot clé ***this()*** et transmettez-lui les paramètres nécessaires.
Si on omet cet appel, l'exécution du constructeur est de toute façon précédée par l'exécution du constructeur sans paramètre de la classe de base.
Si une classe n'a aucun constructeur explicitement défini, Java définit le constructeur par défaut, qui est sans paramètre, et qui, pour une classe héritière, consiste en l'appel du constructeur sans paramètre de la classe de base.

Exemples:

```
class Personne {
String nom, prenom;
int age;
String adresse;
int nombreEnfants;
```

```
        Personne(String n, String p, int a) {
             nom=n; prenom=p; age=a ; nombreEnfants=0;
        }
        Personne(String n, String p, int a, String adresse){
            this(n, p, a); // appel au constructeur de la classe
            this.adresse=adresse;
        }
}
class Salarie extends Personne {
int salaire;
        int prime() {
             return 5*salaire*nombreEnfants/100;
        }
        Salarie (String n, String p, int a, int s) {
            super(n, p, a); // appel du constructeur de la super classe
            salaire=s;
        }
}
```

Voici un autre exemple simple avec deux classes (Parent et Enfant).

```
class Parent {
int x = 1;
   int uneMéthode() {
        return x;
   }
}
class Enfant extends Parent {
int x; // ce x fait partie de cette classe
   int uneMéthode() { // ceci redéfinit la méthode de la classe parent
        x = super.x + 1; // accès au x de la classe parent avec super
        return super.uneMéthode() + x;
   }
}
```

3.4 Accessibilité

En ce qui concerne l'accessibilité des composants d'une classe de base à partir des textes des classes dérivées, le langage offre les quatre modes suivants :

- ✓ ***aucun attribut*** : accessibles par les classes qui font partie du même package, inaccessibles par les autres.

- ✓ **public** : accessibles par toutes les classes.
- ✓ **protected** : accessibles par toutes les classes ***dérivées***, et les classes du même package, inaccessibles par les autres.
- ✓ **private** : inaccessibles par toutes les classes.

L'attribut **protected** permet de rendre accessibles certains membres pour la conception d'une classe dérivée mais inaccessibles pour les utilisateurs de la classe de base. Comme le montre l'exemple suivant, l'accès à un composant **protected** est interdit en situation d'***utilisation*** de la classe de base : depuis la classe **B**, bien qu'héritière de **A**, l'accès **a.JJ** est interdit, car il s'agit d'une utilisation de **A**.

```
package aa;

public class A {
...
protected int JJ;
...
}
```

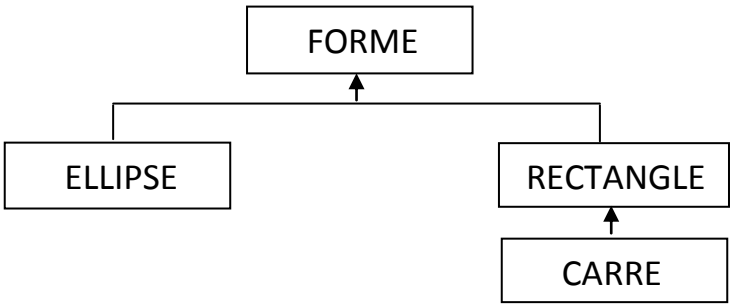
```
package bb;
import aa.*;

class B extends A {
...
void PP() {          autorisé
... JJ++; ... B b; ... b.JJ++;
...
A a; ... a.JJ++; ...
}          interdit
}

class C {
...
void QQ() {A a; ... a.JJ++; ...}
}                  interdit
```

3.5 Compatibilité entre types

Le fait qu'un type **B** hérite d'un type **A** signifie que **B** est un ***sous-type*** de **A**, c'est-à-dire qu'un objet de type **B** ***est*** également de type **A**.

Exemple :

Une variable de type Rectangle désigne des objets de type Rectangle, mais pourra également désigner des objets de type Carré. Si la variable R est de type Rectangle et la variable C est de type Carré, l'affectation R← C est valide car un Carré est un Rectangle dont la longueur et la largeur sont égales. Par contre un Rectangle n'est pas nécessairement un Carré C←R est invalide.



Autre exemple, un **Salarié** est également une **Personne**. Donc toute opération applicable au type **A** est également applicable au type **B**.

Le type **B** est dit ***plus précis*** que **A**, et **A *plus vague*** que **B**.

La règle générale de compatibilité de types peut être informellement énoncée ainsi:

Partout où une expression de type **A** est attendue, une expression de type plus précis que **A** est acceptable. En revanche, l'inverse est interdit.

Cette règle s'applique essentiellement en deux circonstances : en partie droite d'une affectation ou bien en tant que paramètre effectif d'une fonction.

Par exemple, avec les déclarations suivantes :

```
Personne p ...;
Salarié s ...;
void R(Personne p) { ... };
```

Ces instructions sont permises :

```
p = s;
R(s);
```

Pendant l'exécution, la variable **p** et le paramètre formel **p** sont des références sur l'objet précis de type **Salarie** qui leur est assigné.
Cependant **p** ne donne pas directement accès aux membres du type précis **Salarie** : on ne peut pas écrire **p.salaire**, ni **p.prime()**. Le compilateur refuse ces expressions, à juste titre car l'objet désigné par **p** pourrait être parfois du type **Personne**, et ces accès n'auraient aucune signification.

Pour profiter vraiment du fait que l'objet désigné par **p** est du type précis **Salarie**, il faut utiliser la notion de ***méthode virtuelle (ou abstraite)*** décrite au paragraphe suivant.

3.6 Classes abstraites

Une classe abstraite est une classe très générale qui décrit des propriétés qui ne seront définies que par des classes héritières, soit parce qu'elle ne sait pas comment le faire, soit parce qu'elle désire proposer différentes mises en œuvre (Voir exemple classe Personne ci-dessous).

Quand une classe contient une méthode abstraite, la totalité de la classe doit être déclarée comme abstraite. Cela signifie qu'une classe qui inclut au moins une méthode abstraite **ne peut pas être instanciée,** il n'est pas possible de créer des objets de type de la classe abstraite. De plus les classes héritières sont dans l'obligation de redéfinir les méthodes de la classe abstraite sinon elles seront considérées elles mêmes comme abstraites et ne pourront donc pas être instanciées.

Pour declarer une méthode ou une classe abstraite, il faut rajouter le mot clé abstract comme suit:

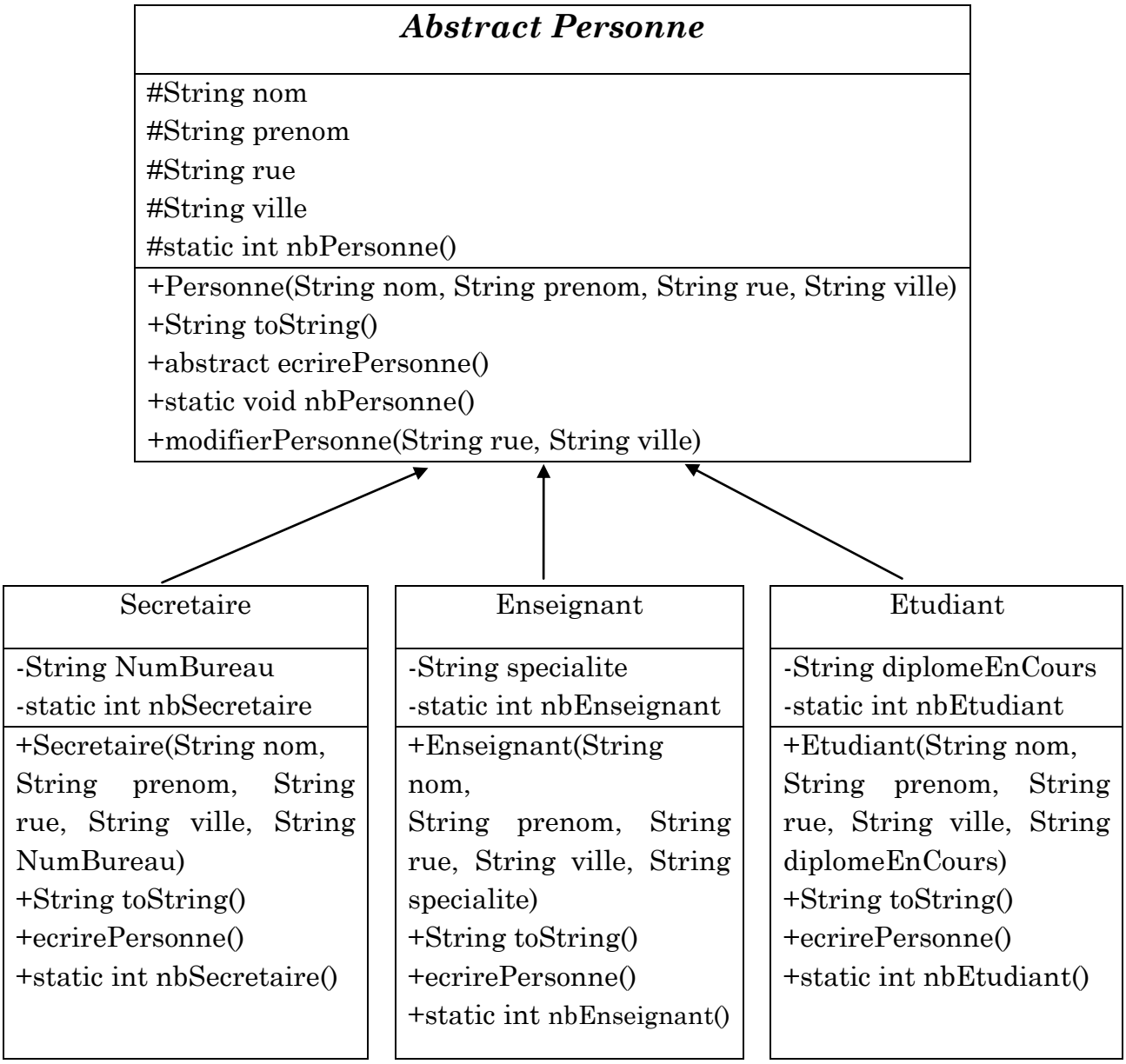
```
abstract class A {
…
abstract void P() ; // Méthode virtuelle
…
void Q() { … }
}
```

```
class B extends A {
…
    void P() { … } // redéfinie
…
void Q() { … }
}
```

Résumé :

- Le mot clé ***abstract*** déclare une méthode ou une classe abstraite.
- Une méthode abstraite est une méthode dont on fournit la déclaration mais pas l'implémentation (c.-à-d. le code).
- Toute classe ayant au moins une méthode abstraite devient abstraite et doit être déclarée comme telle.
- Il est interdit de créer une instance de classe abstraite – pas de new – mais il est possible de déclarer et manipuler des objets du type de telle classe.

Exemple :



 # indique un attribut protected, - indique un attribut private, + indique un attribut public

a/ La classe abstraite Personne :

Si dans l'établissement, il n'y a que 3 catégories de personnes, une personne n'existe pas. Seuls existent des secrétaires, des enseignants et des étudiants. Néanmoins, la classe Personne est utile dans la mesure où elle permet de regrouper les attributs et les méthodes communs aux 3 catégories de personnel.

La classe Personne est déclarée abstraite. On ne peut pas créer (instancier) d'objets de type Personne (Personne p=new Personne("Abed", "Lyes", "rue des rosiers", "Alger") ; fournit une erreur à la compilation.

b/ Les classes Secrétaire, Enseignant et Etudiant :

Une secrétaire est une personne. Elle possède les caractéristiques d'une personne (nom, prenom, rue, ville) plus les caractéristiques spécifiques d'une secrétaire. Il en est de même pour les deux autres classes.

Les méthodes suivantes sont définies pour un objet de la classe Secretaire :

- Secretaire (Strind nom, string prenom, String rue, String ville, String numeroBureau) ; le constructeur d'un objet de la classe Secretaire doit fournir les caractéristiques pour construire une personne, plus les spécificités de la classe Secretaire (numéro de bureau).
- String toString() ; fournit une chaîne contenant les caractéristiques d'une Secretaire.
- ecrirePersonne() ; écrit "Secrétaire :" suivi des caractéristiques d'une secrétaire.

De même, un Enseignant est une Personne enseignant une spécialité. Un Etudiant est une Personne préparant un diplôme. Les méthodes pour Enseignant et Etudiant sont similaires à celles de Secretaire. Une variable privée static dans chaque classe compte le nombre de personnes créées dans chaque catégorie. Une méthode static du même nom que la variable fournit la valeur de cette variable static (nbSecretaire, nbEnseignant, nbEtudiant).

```
public abstract class Personne {
protected String nom;
protected String prenom;
protected String rue;
protected String ville;
protected static int nbPersonne=0;

        public Personne( String nom,String prenom, String rue,String ville) {
          this.nom=nom;
          this.prenom=prenom;
          this.rue=rue;
          this.ville=ville;
          nbPersonne++; /* est une variable static qui comptabilise le nombre de
                                    personne dans l'établissement*/
        }

        public String toString(){
                return nom+"  "+prenom+"  "+rue+"  "+ville ;
        }

        abstract void ecrirePersonne(); /* Méthode abstraite à redéfinir dans les
                                                  classes dérivées */

        public static void nbPersonne(){
           System.out.println("Nombre de personnes:" + nbPersonne +
                        "\nNombre d'enseignants:" +  Enseignant.nbEnseignant()+
                        "\nNombre de secrétaires:"+ Secretaire.nbSecretaire()+
                        "\nNombre d'Etudiants:"+ Etudiant.nbEtudiant());
        }
        public void modifiePersonne(String rue,String ville){
            this.rue=rue;
            this.ville=ville;
            ecrirePersonne(); /* Pour vérifier que la modification a bien été faite */
        }
}

public class Secretaire extends Personne {

private String numBureau;
private static int nbSecretaire=0;

        public Secretaire(String nom, String prenom, String rue, String ville,
                                String numBureau) {
                super(nom, prenom, rue, ville);
                 this.numBureau=numBureau;
                nbSecretaire++;
        }
```

```
        public String toString(){
                return super.toString()+"\nNumBureau: "+numBureau;
        }
        public void ecrirePersonne(){
                System.out.println("Secretaire: "+ toString());
        }
        public static int nbSecretaire(){
                return nbSecretaire;
        }
}

public class Enseignant extends Personne {

        private String specialite;
        private static int nbEnseignant=0;

                public Enseignant(String nom, String prenom, String rue, String
                                        ville, String specialite) {
                        super(nom, prenom, rue, ville);
                        this.specialite=specialite;
                        nbEnseignant++;
                }

                public String toString(){
                        return super.toString()+"\nSpécialité: "+ specialite;
                }
                public void ecrirePersonne(){
                        System.out.println("Enseignant: "+ toString());
                }
                public static int nbEnseignant(){
                        return nbEnseignant;
                }
}

public class Etudiant extends Personne{

        private String diplome;
        private static int nbEtudiant=0;

                public Etudiant(String nom, String prenom, String rue, String
                                        ville, String diplome) {
                        super(nom, prenom, rue, ville);
                        this.diplome=diplome;
                        nbEtudiant++;
                }
```

```
        public String toString(){
            return super.toString()+"\nDiplome: "+ diplome;
        }

        public void ecrirePersonne(){
            System.out.println("Etudiant: "+ toString());
        }
        public static int nbEtudiant(){
            return nbEtudiant;
        }
}

public class PPPersonne {   // classe exécutable

  public static void main(String[] args) {

      Secretaire Sec=new Secretaire("Seba","Zohra","rue des rosiers", "Alger",
                                    "A326");
      Sec.ecrirePersonne();

      Enseignant Ens=new Enseignant ("Fridi","Boualem","rue des mimosas",
                                     "Rouiba", "Mathematique");
      Ens.ecrirePersonne();

      Etudiant Etu=new Etudiant("Smati","Ryad","rue des lilas", "Alger",
                                "Informatique");
      Etu.ecrirePersonne();

      Personne.nbPersonne();

      System.out.println("\nAprès modification:\n");

      Sec.modifiePersonne("rue des orangers", "BabEzzouar");
      Ens.modifiePersonne("rue des marguerites", "Kouba");

  }
}
```

3.7 Le Polymorphisme

Le polymorphisme (plusieurs formes) est contrôlé par l'héritage. C'est la possibilité pour deux classes séparées, mais reliées, de recevoir le même message mais d'agir dessus de différentes façons. En d'autres termes, deux classes différentes (mais reliées) peuvent avoir la même méthode mais l'implémenter de façon différente.
Vous pouvez ainsi avoir une méthode dans une classe, également implémentée dans une classe enfant, et accéder au code depuis la classe parent (ce qui est similaire au chaînage automatique du constructeur déjà évoqué). Tout comme dans l'exemple du constructeur, le mot clé **super** permettra d'accéder à toutes les méthodes ou variables membre de la classe de niveau supérieur.
Autrement dit le polymorphisme permet de définir dans la super-classe des méthodes tenant compte des spécificités des classes dérivées. En cas de redéfinition de méthodes, c'est la méthode de la classe dérivée de l'objet qui est prise en compte, et non celle de la super-classe. Il faut comprendre qu'une méthode peut se comporter différemment suivant l'objet sur lequel elle est appliquée.

3.7.1 Le polymorphisme d'une méthode redéfinie

Exemple : La méthode ecrirePersonne()

La méthode modifierPersonne modifie l'adresse (rue, ville) d'une personne quelle que soit sa catégorie. Elle se trouve dans la classe Personne. Elle peut être appelée pour un objet Secretaire, Enseignant ou Etudiant.

```
public void modifiePersonne(String rue,String ville){
    this.rue=rue;
    this.ville=ville;
    ecrirePersonne(); /* Pour vérifier que la modification a bien été faite */
}
```

Par contre, que fait la méthode ecrirePersonne() appelé dans modifiePersonne() ?
Le compilateur faisant une liaison statique, il générerait au moment de la compilation des instructions prenant en compte la méthode ecrirePersonne() de la classe Personne et qui en fait ne fait rien sur cet exemple.
Quand le compilateur prévoit une liaison dynamique, la méthode ecrirePersonne() prise en compte dépend à l'exécution de la classe de l'objet. La méthode est d'abord recherchée dans la classe de l'objet, et en cas d'échec dans sa super-classe.

Si chaque sous-classe redéfinit la méthode, celle de la super-classe n'est jamais exécutée et peut être déclarée abstraite. Elle sert de prototype lors de la compilation de modifiePersonne() de la classe Personne. Si la méthode est déclarée abstraite, les sous-classes doivent la redéfinir sinon, il y a un message d'erreur à la compilation.

Les instructions suivantes appellent la méthode modifiePersonne() de Personne pour deux objets différents dont la classe est dérivée de Personne.

```
Sec.modifiePersonne("rue des orangers", "BabEzzouar");
Ens.modifiePersonne("rue des marguerites", "Kouba");
```

Pour l'objet Sec la méthode modifiePersonne() appelle la méthode ecrirePersonne() de Secretaire. Pour l'objet Ens la méthode modifiePersonne() appelle la méthode ecrirePersonne de Enseignant. On parle alors de polymorphisme. Le même appel de la méthode ecrirePersonne() de modifiePersonne déclenche des méthodes redéfinies différentes (portant le même nom) de différentes classes dérivées.
Le polymorphisme découle de la notion de redéfinition des méthodes entre une super-classe et une sous-classe.

3.7.2 Le polymorphisme d'une variable de la super-classe

Le programme PPPersonne vu ci-dessus pourrait être écrit comme suit. La variable locale p est une référence sur une Personne. On ne peut pas créer d'instance de Personne (la classe est abstraite), mais on peut référencer à l'aide d'une variable de type Personne, un descendant de Personne (Secretaire, Enseignant ou Etudiant). L'instruction p.ecrirePersonne() est polymorphe ; la méthode appelée est celle de la classe de l'objet.

```
public class PPPersonne2 {   // classe exécutable

  public static void main(String[] args) {

      Personne p ;
      p=new Secretaire("Seba","Zohra","rue des rosiers", "Alger",
                                          "A326");
      p.ecrirePersonne();// ecrirePersonne()  de Secretaire

      p=new Enseignant ("Fridi","Boualem","rue des mimosas",
                                          "Rouiba", "Mathematique");
      p.ecrirePersonne(); // ecrirePersonne()  de enseignant
```

```
        p=new Etudiant("Smati","Ryad","rue des lilas", "Alger",
                                        "Informatique");
        p.ecrirePersonne(); // ecrirePersonne() de Etudiant

        p=new Etudiant("Zettel","Ghania","rue des hortensias", "Elbiar",
                                        "Biologie");
        p.ecrirePersonne(); // ecrirePersonne() de Etudiant

        Personne.nbPersonne();  // Méthode static
```

Peut-on à partir d'une référence de Personne, accéder aux attributs ou aux méthodes non redéfinies d'une classe dérivée ? Réponse : oui en forçant le type si l'objet est bien du type dans lequel on veut le convertir (sinon, il y a lancement d'une exception de type ClassCastException). (Voir chapitre Exceptions)

```
        // A partir d'une référence de Personne,
        //accéder à un objet dérivé

        // p référence le dernier Etudiant créé ci-dessus.
        // Erreur de compilation : diplomeEncours n'est pas une méthode
        // de la classe Personne
        // System.out.println("Diplôme en cours :" + p.diplomeEncours()) ;

        // Erreur de compilation : impossible de convertir Personne
        // en Secretaire implicitement
        // Secretaire Sec=p ;

        // Erreur d'exécution : Exception ClassCastException :
        // p n'est pas une Secretaire (c'est un Etudiant)
        // Secretaire Sec=(Secretaire)p ;

        Etudiant Etu1=(Etudiant)p ;  // OK p est un Etudiant
        System.out.println("Diplôme en cours :" + Etu1.diplomeEncours()) ;

        // teste si p est un objet de la classe Etudiant
        if (p instanceof Etudiant) {
           System.out.println("p est de la classe Etudiant") ;
        }
    } // fin main
} // fin PPersonne2
```

3.7.3 Le polymorphisme de toString()

Si on remplace la méthode abstraite de la classe Personne par :

```
void ecrirePersonne() {
  System.out.println(toString()) ; // ou System.out.println (this);
}
```

Et si on supprime ecrirePersonne() dans les sous-classes Secretaire, Enseignant et Etudiant, on obtient le résultat d'exécution ci-dessous.
System.out.println(this) ; est une écriture simplifiée à System.out.println (toString()).

La méthode ecrirePersonne() appelée est celle unique de la classe Personne ; par contre la méthode toString() appelée dans ecrirePersonne() est celle de la classe de l'objet (Secretaire, Enseignant ou Etudiant) en raison de la liaison dynamique. Si plus tard, on définit une classe Technicien, la méthode ecrirePersonne() de Personne fera appel à la méthode toString() de Technicien, et ce sans aucune modification. Il suffira de définir une méthode toString() pour la classe Technicien. En l'absence de cette définition, c'est la méthode toString() de Personne qui sera utilisée.
La méthode toString() de Personne est polymorphe puisque le même appel de méthode met en œuvre à l'exécution des méthodes dérivées redéfinies différentes suivant l'objet qui déclenche l'appel.

3.7.4 Le modificateur « final »

Dans la définition d'une variable de classe ou d'instance, le modificateur indique que la variable ne pourra changer de valeur et représente une constante.
On peut définir une classe ou méthode avec ce modificateur et l'on obtient les caractéristiques suivantes :

- Une méthode définie « final » dans une classe *CC* ne peut être redéfinie dans une sous classe de *CC*.
- Une classe définie final ne peut pas être héritée. (final class A { … })

Pour une méthode ou une classe, on renonce ainsi à l'intérêt, pourtant souvent majeur de l'héritage. Si on opte parfois pour cette solution, c'est essentiellement pour des raisons de sécurité ou de performance.

3.7.5 La super-classe OBJECT

En java, toutes les classes dérivent implicitement d'une classe Object qui est la racine de l'arbre d'héritage. Cette classe définit une méthode toString() qui écrit le nom de la classe et un numéro d'objet. Les classes dérivées peuvent redéfinir cette méthode. Puisque toute classe dérive de la classe Object, une référence de type Object peut référencer n'importe quel type d'objet puisqu'une référence de la super-classe peut être utilisée pour les classes dérivées. (voir polymorphisme d'une variable).

```
Object objet ;
objet=new Secretaire("Seba","Zohra","rue des rosiers", "Alger",
                                                    "A326");
objet= new String("Bonjour") ;
objet= new Complex (2, 1.50) ;
```

3.8 Les Interfaces

L'héritage multiple n'est pas permis en JAVA. Cependant Java offre la notion d'***interface***, notion voisine, plus simple et suffisante dans les cas usuels.
Une interface permet de regrouper des classes ayant une ou plusieurs propriétés communes et de déclencher des traitements pour les objets des classes implémentant cette interface.
Une interface est une collection de déclarations de méthodes. On peut la considérer comme un cas limite de classe abstraite : elle ne définit aucun corps de méthode, ni aucun composant variable. Il n'y figure que le profil des méthodes et éventuellement des déclarations de constantes (ni d'attributs ni d'implémentation de méthodes).

En plus de l'héritage simple (mot clé **extends**), une classe peut "***implémenter***" une ou ***plusieurs*** interfaces (mot clé **implements**), en définissant les méthodes de ces interfaces.

Une classe ne peut hériter que d'une seule super-classe mais peut implémenter plusieurs interfaces.

Une interface s'utilise comme une classe abstraite. Étant donné une interface ***I***, on peut déclarer des variables, des résultats ou des paramètres de type ***I***. Tout objet instance d'une classe qui implémente cette interface est compatible avec ces variables, résultats ou paramètres.

```
interface nom_interface {
<déclaration de varaibles> ;
<déclaration de méthodes> ;
}
class nom_classe extends classe_base implements interface1, interface2, .. .  {
// …
}
```

Une classe qui n'implémente pas toutes les méthodes définies dans l'interface ne peut pas être instanciée.

Résumé :

- Une interface ne contient que des méthodes qui sont par définition abstraites.
- Elle permet le partage de comportements entre des classes qui n'ont pas de lien hiérarchique.

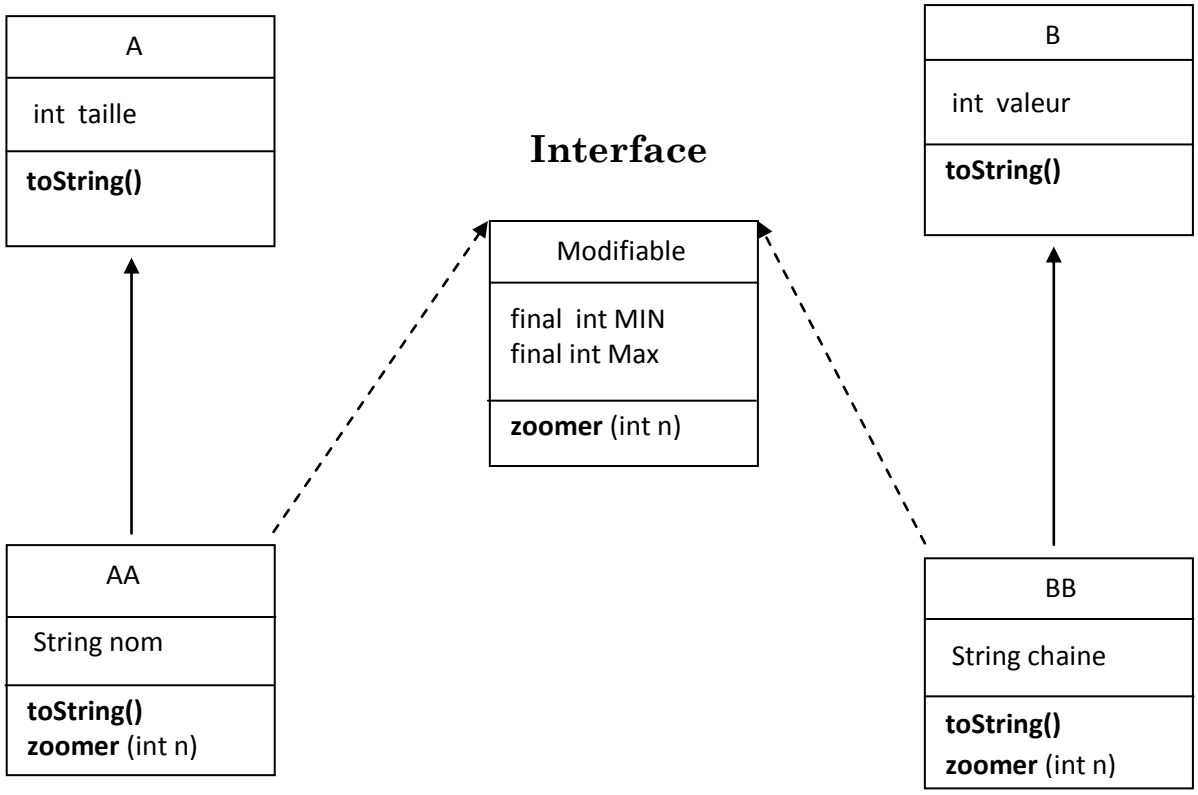
- Toute classe qui déclare implémenter une interface doit fournir le code correspondant aux méthodes de cette interface.
- Une interface définit un ensemble de méthodes mais ne les implémente pas.

Une interface se différencie d'une classe abstraite :

- Aucune méthode ne peut être implantée dans une interface.
- Une classe peut implémenter plusieurs interfaces.
- Une interface est en dehors de la hiérarchie des classes.

Remarque : Une interface ne devrait pas croître par ajout de nouvelles méthodes, car alors toutes les implantations déjà créées ne respecteraient plus le contrat. Il est donc nécessaire, dans le principe, de spécifier une interface de façon complète dès sa création.

Exemple :



Le programme java suivant permet de définir des variables de type Modifiable référençant des AA ou des BB, et de déclencher la méthode zoomer() pour des Modifiable.

```
class A {
  int taille ;
        A (int taille) {
             this.taille=taille ;
        }
// String des caractéristiques de A
        public String toString() {
             return "A : "+ taille ;
        }
} // fin A

// ----------------    AA hérite de A  ----------------------------------------
class AA extends A implements Modifiable {
String nom ;

        AA (String nom, int taille) {
             super(taille) ;
             this .nom=nom ;
        }
        void incrementer () {
               taille++ ;
        }
        public String toString() {
             return "AA : "+ super.toString() + ", AA : " + nom ;
        }
        public void zommer(int n) { // méthode de l'interface redéfinie pour AA
             if ( n<MIN) n=MIN ;
             if (n>MAX) n=Max;
             if (n==0) n=1;
            // n>0, on multiplie par n
           // n<0, on divise par |n|
           taille=n>0 ? taille*n : (int) (taille / (double) –n) ;
        }
}// fin AA
//-------------------------------------------------------------------------
class B {
   int numero ;

        B (int numero) {
              this.numero=numero ;
        }
        // String des caractéristiques de B
        public String toString ()  {
             return "B : "+ numero;
        }
}// fin B
```

```
// ---------- BB hérite de B et implémente l'interface Modifiable ----------

class BB extends B implements Modifiable {
  String chaine ;

        BB (String chaine, int numero) {
              super(numero);
              this.chaine=chaine;
        }

        // String des caractéristiques de BB
        public String toString() {
              return "BB : "+ super.toString() + " , BB : " + chaine ;
        }

        // Méthode de l'interface Modifiable redéfinie pour BB
        public void zoomer(int n) {
                  if (n<MIN) n=MIN ;
                  if (n>MAX) n=MAX ;
                  if (n==0)  n=1;
                 // n>0, on multiplie par 2*n
                 //n<0, on divise par 2*|n|
                numero=n>0 ? numero*2*n : (int) (numero/ (double) (-2*n)) ;
        }
} // fin BB
```

```
interface Modifiable {

   static final int MIN=-5 ;
   static final int MAX=+5 ;

  void zoomer(int n);
} // fin Modifiable
```

Chacune des classes désirant être reconnues comme ayant la propriété Modifiable doit le faire savoir en implémentant l'interface Modifiable,

class AA extends A implements Modifiable { …}

class BB extends B implements Modifiable { …}

et en définissant les méthodes données comme prototype dans l'interface (soit la seule méthode void zoomer(int) sur l'exemple).

On peut définir des variables de type Modifiable (On ne peut pas instancier d'objet de type Modifiable) référençant des objets des classes implémentant l'interface.

Une variable Modifiable peut référencer un objet AA ou un objet BB. On peut même faire un tableau de Modifiable contenant des objets implémentant l'interface Modifiable (AA ou BB). Une variable de type Modifiable ne donne accès qu'aux méthodes de l'interface. Pour accéder à une des méthodes d'un objet AA à partir d'une variable de type Modifiable, il faut forcer le type (cast) en AA. Si le transtypage échoue (l'objet n'est pas un AA), il y a lancement d'une exception de type ClassCastException.

```
public class PPInterface  { // programme principal

        public static void main(String[ ] args) {
           // un tableau de Modifiable contenant des AA ou des BB
           Modifiable[ ] tabMod = { new AA ("b1", 10), new AA ("b2", 20),
                                    new BB ("b3", 30), new BB ("b4", 50)
                                  } ;

            // On déclenche la méthode zoomer() pour les éléments du tableau
            for(int i=0 ;i<tabMod.length ; i++) {
                  tabMod[i].zoomer(-2);
            }
            for(int i=0 ;i<tabMod.length ; i++) {
                  // System.out.println(tabMod[i].toString());
                   System.out.println(tabMod[i]);
            }

            // erreur : un modifiable ne peut pas accéder à incrementer()
           for(int i=0 ;i<tabMod.length ; i++) {
                // erreur : incrementer n'est pas une méthode de la classe Modifiable
                // tabMod[i].incrementer() ;
                System.out.println(tabMod[i].getClass().getName());
            }

            Modifiable m1= tabMod[0];
            ((AA)m1).incrementer(); // il faut effectuer un transtypage
            System.out.println("m1 : "+m1) ;

            Modifiable m2= tabMod[2];  //m2 est de type BB
            ((AA)m2).incrementer();     //Exception ClassCastException
            System.out.println("m2 : "+m2) ;
        } // fin main

} // fin PPInterface
```

4. Les exceptions: *throw-try-catch*

L'exécution anormale d'une action peut provoquer une erreur de fonctionnement d'un programme.
Jusqu'à présent, lorsqu'une situation anormale était détectée, le programme signalait le problème par un message d'erreur et s'arrêtait. Cette façon est brutale et expéditive. Dans certains cas, il serait souhaitable qu'il reprenne le contrôle afin de poursuivre son exécution. Les exceptions offrent une solution à ce problème.
L'origine d'une exception est très diverse. Il peut s'agir d'exception matérielle (défectueux), ou d'une allocation mémoire impossible car l'espace mémoire est insuffisant. Ce type d'exception n'est pas directement de la responsabilité du programmeur (ou du programme).
En revanche, les exceptions logicielles le sont, comme par exemple, une division par zéro ou l'indexation d'un composant de tableau en dehors du domaine de définition du type des indices

4.1 Le mécanisme d'exception de java

Une exception est décrite par un objet, occurrence d'une sous-classe de la classe ***Throwable***. Cette classe possède 2 sous-classes directes :

- La classe ***Error***, décrit des erreurs systèmes comme par exemple l'absence de mémoire. Ces exceptions ne sont en général pas traiter par le programme.
- La classe ***Exception***, décrit des exceptions logicielles.

4.2 Traitement d'une exception

En résumé, la notion d'exception permet de traiter de manière plus souple et plus lisible les cas exceptionnels, essentiellement les cas d'erreurs.
Une exception est donc un événement exceptionnel risquant de compromettre le bon déroulement du programme. Un débordement de tableau ou de pile, la lecture d'une donnée erronée ou d'une fin de fichier prématurée constituent des exemples d'exceptions.

En java, la fonction peut lancer une exception en indiquant un type d'exception.
L'exception se « propage » de retour de méthode en retour de méthode jusqu'à ce qu'une méthode la traite. Si aucune méthode ne la prend en compte, il y a un message d'erreur d'exécution. Suite à une exception, toutes les instructions sont ignorées, sauf les instructions chargées de capter cette exception.

Pour traiter une exception produite par l'exécution d'une action A, il faut placer la méthode dans une clause **try**, suivie obligatoirement d'une clause **catch** qui contient le traitement de l'exception.

```
…
try {
// Instructions susceptibles de provoquer des erreurs;
    A
}
catch (TypeException e) {
// Instructions de traitement de l'erreur;
   B
}
// Instructions si aucune erreur est apparue;
…
```

Si l'action **A** contenue dans **try** détecte une anomalie qui émet une exception de type ***TypeException***, son exécution est interrompue. Le programme se poursuit par l'exécution de **B** placée dans **catch.**

Si aucune situation anormale n'a été détectée, l'action **A** est exécutée et l'action **B** ne l'est pas.

Exemple :

```
public int lireEntier() {
        try {
                return Sc.nextInt() ;
        }
        catch (InputMismatchException  e) {
                System.out.println ("Réessayez:"+e) ;
                return lireEntier();
        }
}
```

Cette méthode contrôle la validité d'une valeur entière lue au clavier. Si la lecture produit l'exception ***InputMismatchException*** (la valeur lue n'est pas un entier) la clause ***catch*** propose une nouvelle lecture (Remarque : le nombre de lecture possible n'est pas borné).

Plusieurs clauses ***catch***, associées à des exceptions différentes, peuvent suivre une clause ***try***.

Quelques types d'exceptions :

IOException : exception liée aux entrées/sorties
IndexOutOfBoundsException : accès hors des bornes d'un tableau
NullPointerException : accès à travers null.

Exemple2 : *Le débordement de tableaux (dans main)*

```
public class TestException1 {
     public static void main(String[] args) {
       int tabEnt[]={1, 2, 3, 4};
      // Exception de type java.land.ArrayIndexOutOfBoundsException
     // non contrôlée par le programme ; erreur d'exécution
    tabEnt[4]=0 ; // erreur et arrêt
   System.out.println("Fin du main") ;   // le message n'apparait pas
}  // main
} // class TestException
```

Dans le programme suivant, la même erreur d'accès à un élément est captée par le programme. Lors de la rencontre de l'erreur, les instructions en cours sont abandonnées et ***un bloc de traitement (catch) de l'exception est recherché et exécuté.*** Le traitement se poursuit après ce bloc sauf indication contraire du catch (return ou arrêt par exemple). Le message « Fin du main » est écrit après interception de l'exception.

```
public class TestException2 {
     public static void main(String[] args) {
       try {    int tabEnt[]={1, 2, 3, 4};
        // Exception de type java.land.ArrayIndexOutOfBoundsException
       // contrôlée par le programme
        tabEnt[4]=0 ;       // Exception et suite au catch
        System.out.println("Fin du try") ;   // non exécutée
      } catch (Exception e) {   // l'erreur va être capturée dans l'objet e
            System.out.println("Exception : " + e);   // écrit le message
         }
   System.out.println("Fin du main") ;   // Exécutée
}  // main
} // class TestException
```

Exemple de résultat d'exécution :

```
Exception : java.ArrayIndexOutOfBoundsException

Fin du main
```

4.3 throw et throws

Une méthode qui contient une action susceptible de produire des exceptions n'est pas tenue de la placer dans une clause ***try*** suivie d'une clause ***catch***.
Dans ce cas, elle doit indiquer dans son en-tête, précédé du mot clé « throws Exception », les exceptions qu'elle ne désire pas capturer. Si une exception apparait alors l'exécution se poursuit dans l'environnement d'appel de la méthode.

Exemple 3 :

```
package piles;
import java.util.Scanner;
public class Pile {
        int sommet;
        int tpile[];
        //final int max=3;

        public Pile(int taille)
        {sommet=-1;
         tpile= new int [taille];
        }
        private void erreur(String mes) throws Exception {
                throw new Exception (mes);
       }
        public boolean  pilevide()
        { if (sommet==-1) return true ;
          else return false;
        }
        public boolean pilepleine()
        { if (sommet==tpile.length-1) return true;
          else return false;
        }
        public void empiler(int x) throws Exception {
        if (!pilepleine()) {
                sommet++;  tpile[sommet]=x;
        }
        else {   erreur("Pile saturée");    }
        }
```

```
    public int desempiler() throws Exception {
     int x=0;
     if (!pilevide()) {
            x=tpile[sommet]; sommet--;
     }
     else { erreur("pile vide");}
     return x;
    }
    public int sommetpile()  {
        return (tpile[sommet]);
    }

    // sans try catch il faudra rajouter throws Exception
    public void remplirpile(int n )
    {  int i;
       Scanner E= new Scanner(System.in);
       System.out.println("Donnez les valeurs entières");
       try {  for (i=0;i<n;i++) empiler(E.nextInt());   }
       catch (Exception e) {   System.out.println("Erreur sur pile  "+e);    }
    }
    public  Pile affichpile() throws Exception
    {int x;
      Pile R=new Pile(tpile.length);
       while(!pilevide()){
            x=desempiler();
            R.sommet++; R.tpile[R.sommet]=x;
           System.out.print(x + "  ");
        }
       return R;
     }
}

import java.util.Scanner;
import piles.Pile;
public class PPile {
    public static int menu() {
    Scanner E= new Scanner(System.in);
    System.out.println("\n\nGESTION D'UNE PILE\n" +
                       "0 - Fin \n" +
                       "1 - Initialisation de la pile\n" +
                       "2 - La pile est-elle vide?\n" +
                       "3 - Insertion dans la pile\n" +
                       "4 - Retrait de la pile\n" +
                       "5 - Affichage de la pile\n\n" +
                       "Votre choix ?");
    return E.nextInt();
  }
```

```
public static void main(String[] args) throws Exception {
  Scanner E= new Scanner(System.in);
  int x;
  boolean fini=false;
  Pile pile1=new Pile(3);

  try {
  while (!fini) {
          switch (menu()) {

          case 0: //fin
                  fini=true;
                  break;

          case 1: // initialisation
                  System.out.println("Donnez la taille max de la pile");
                  int taille=E.nextInt(); int n;
                  pile1 = new Pile(taille);

                  System.out.println("donner le nombre d'éléments");

                  try {
                          n=E.nextInt();
                          if (n<taille) pile1.remplirpile(n);
                  }
                  catch (Exception e) {
                          System.out.println("le nombre d'éléments doit être < à
                          la taille de la pile Réessayez");
                  }
                  break;

          case 2: // la pile est-elle vide?
                  if (pile1.pilevide())
                      System.out.println("La pile est vide");
                  else System.out.println("La pile n'est pas vide");
                  break;

          case 3: // empiler une valeur
                  System.out.println("Donnez une valeur a empiler: ");
             x= E.nextInt();
             pile1.empiler(x);
             break;

          case 4: // désempiler une valeur
                  x=pile1.desempiler();
                  System.out.println("Valeur desempiler: "+x);
                  break;
```

```
                    case 5: // Afficher la pile
                    pile1=pile1.affichpile();
                    break;
                }// fin switch
            }// fin while
        }
    catch (Exception e){
        System.out.println("Erreur sur pile " + e);
    }// fin try
     System.out.println("************************************");
    Pile pile2=new Pile(5);
    pile2.remplirpile(3);
    System.out.println("Affiche Pile2");
    pile2=pile2.affichpile();
    System.out.println("Affiche Pile1");
    pile1.affichpile();
}// fin main
}// fin de la classe principale
```

Exemple d'exécution :

```
Votre choix ? 3
Valeur à empiler ? 15
Erreur sur pile : java.lang.Exception : Pile Saturée
```

La hiérarchie des exceptions : Les exceptions constituent un arbre hiérarchique utilisant la notion d'héritage.

Object	la classe racine de toutes les classes
Throwable	getMessage(), toString(), printStackTrace(), etc.
Error	Erreur grave, abandon du programme
Exception	anomalies récupérables
RunTimeException	erreur à l'exécution
ArithmeticException	division par zéro par exemple
ClassCastException	mauvaise conversion de classes (cast)
IndexOutOfBoundsException	en dehors des limites (tableau)
NullPointerException	référence nulle
IllegalArgumentException	
NumberFormatException	nombre mal formé
IOException	erreurs d'entrées-sorties
FileNotFoundException	fichier inconnu

Dans un catch, on peut capter l'exception à son niveau le plus général (Exception), ou à un niveau plus bas (FileNotFoundException par exemple) de façon à avoir un message plus approprié. On peut avoir plusieurs catch pour un même try.

5. Généricité

La généricité est la possibilité de paramétrer la définition de modules logiciels. Un cas fréquent et très utile de généricité consiste à définir des types paramétrés par d'autres types. Par exemple, on peut ainsi définir le type Pile(T), les piles d'éléments de type quelconque T. Une telle définition s'appelle un ***type générique***. On peut ensuite utiliser ce type générique pour disposer des piles d'entiers, Pile (int), de caractères, Pile(char), ou de tout autre type passé en paramètre effectif.

Java permet de les réaliser, en respectant certaines conventions, au moyen de la classe Object et en pratiquant le typage forcé (cast).
La classe Object est la classe dont héritent implicitement toutes les classes programmées en java. Ainsi pour structurer des données de type quelconque, il suffit de structurer des objets de type Object. Par exemple, la classe pile d'éléments de n'importe quel type se programme ainsi :

```
public class Pile {
        private int sommet;
        private Object  tpile[];
        //final int max=3;

        public Pile(int taille)
        {sommet=-1;
         tpile= new Object [taille];
        }

        public void empiler(Object  x) {
          …
        }
       …
```

6. Les collections

Les collections en Java sont des classes permettant de manipuler les structures de données usuelles : listes, vecteurs, files, piles, etc. Une manière standard de représenter une structure de donnée en Java consiste à regrouper dans une interface l'ensemble des noms des opérations applicables sur celle-ci (ajout, suppression, effacement, etc.), c'est-à-dire l'ensemble des opérations mentionnées dans le type de données abstrait implémenté par cette structure (sa spécification, indépendamment du choix d'implémentation). Cette interface sera implémentée par chaque classe représentant cette sorte de structure : par exemple, l'interface *List* regroupe un ensemble de noms de méthodes génériques permettant la manipulation de listes, et est implémentée à la fois par les classes concrètes *ArrayList* (implémentation des listes par tableaux extensibles) et LinkedList (implémentation des listes par chaînage).

Les interfaces et classes citées ci-dessous se trouvent dans le package java.util.

Voici quelques classes :

- ***LinkedList*** fournit une liste doublement chaînée. L'ajout d'un élément est rapide, mais la recherche d'une valeur donnée et l'accès au $i^{ème}$ élément sont en *O(n)*.
- ***ArrayList*** est une implémentation à base de tableau. L'ajout d'un élément peut être plus coûteux que dans le cas d'une LinkedList si le tableau doit grossir, mais par contre l'accès au $i^{ème}$ élément en temps constant.
- ***Vector*** est un équivalent de ArrayList datant du jdk1.0. Les différences entre les deux se situent surtout lorsque l'on utilise le multitâche.
- ***Stack*** est un type spécial de vecteur, utilisé pour réaliser des piles.

6.1 La classe *LinkedList (*Listes chaînées)

Cette classe implémente l'interface List en chaînant les éléments

void addFirst(Object o)	Ajoute un élément en début de liste.
void addLast(Object o)	Ajoute un élément en fin de liste.
Object getFirst()	Retourne l'élément en début de liste.
Object getLast()	Retourne l'élément en fin de liste.
Object removeFirst()	Supprime et retourne l'élément en début de liste.
Object removeLast()	Supprime et retourne l'élément en fin de liste.

6.2 La classe ArrayList

Les listes sont des objets de type ArrayList, un type prédéfini du langage java. La gestion des listes est assez similaire à la gestion d'un tableau, puisque le programme crée une liste avec ajout de données au fur et à mesure des besoins de l'utilisateur. Les données sont enregistrées dans leur ordre d'arrivée. Un indice géré par l'interpréteur permet de retrouver l'information.
Les données enregistrées dans une ArrayList sont en réalité rangées dans un tableau interne créé par l'interpréteur. La taille du tableau interne est gérée automatiquement par java. Ainsi lorsque la liste des éléments à ajouter dépasse la taille du tableau interne, un nouveau tableau est créé et les anciennes valeurs y sont copiées.

Manipulation d'une liste :

Pour utiliser une liste, il est nécessaire de la déclarer de la façon suivante :

```
ArrayList liste = new ArrayList() ;
```

Ainsi déclaré, liste est un objet de type ArrayList, auquel on peut appliquer les méthodes de la classe ArrayList. Ces méthodes, décrites dans le tableau ci-après, permettent l'ajout, la suppression ou la modification d'une donnée dans la liste.

Remarque : Il est possible de construire une liste en précisant la capacité de stockage du tableau interne. De cette façon, on évite la répétition des opérations de création de tableaux internes et la recopie des valeurs qui sont des opérations coûteuses en temps d'exécution. La création d'une liste de taille connue s'effectue en indiquant au constructeur le nombre estimé d'éléments contenu dans la liste :

```
int capaciteInitiale = 20 ;
ArrayListe list = new ArrayList(capaciteInitiale) ;
```

add(objet)	Ajoute un élément objet en fin de liste.
add(indice, objet)	Insère un élément objet dans la liste, à l'indice spécifié en paramètre.
get(indice)	Retourne l'élément stocké à l'indice spécifié en paramètre
clear()	Supprime tous les éléments de la liste
indexOf(objet)	Retourne l'indice dans la liste du premier objet donné en paramètre, ou -1 si objet n'existe pas dans la liste.
lastIndexOf()	Retourne l'indice dans la liste du dernier objet donné en paramètre, ou -1 si l'objet n'existe pas dans la liste
removeRange(indice)	Supprime l'objet dont l'indice est spécifié en paramètre.

removeRange(i, j)	Supprime tous les éléments compris entre les indices i (valeur comprise) et j (valeur non comprise).
set(objet, i)	Remplace l'élément situé en position i par l'objet spécifié en paramètre.
Size()	Retourne le nombre d'éléments placés dans la liste.

Exemple :

```
// Créer un nombre indéterminé d'étudiants
public class Etudiant {
private String nom, prenom ;
private String specialite;
private Date date_naissance;

public Etudiant (String nom, String prenom, String specialite, Date
date_naissance) {
        this.nom = nom;
        this.prenom = prenom;
        this.specialite = specialite;
        this.date_naissance = date_naissance;
}
public void afficheUnEtudiant() {
  System.out.println("Nom : " + nom + "Prenom : " + prenom + "Specialite : " +
         specialite + "date de naissance : ") ; date_naissance.afficheDate() ;
}
} // fin Etudiant

import java.util.*
public class Classe {
private ArrayList liste ;
public Classe() {
  liste = new ArrayList() ;
}
public void ajouteUnEtudiant() {
  liste.add( new Etudiant() ) ;
}
public void afficheLesEtudiants() {
  int nbEtudiants = liste.size() ;
  if (nbEtudiants >0)  {   Etudiant  tmp ;
          for (int i=0 ; i<nbEtudiants ;i++)  {
          tmp=(Etudiant) liste.get(i);  ← cast
          tmp.afficheUnEtudiant();
     }
   }  else System.out.println("Il n' y a pas d'étudiant dans cette liste") ;
}
} // fin de Classe
```

Remarques :

- Une liste a la capacité de mémoriser n'importe quel type d'objet, prédéfini ou non. Il est donc nécessaire d'indiquer au compilateur à quel type correspond l'objet extrait, comme dans tmp=(Etudiant) liste.get(i);
- L'ajout d'un élément dans une liste n'est possible que si l'élément est un objet. Il n'est en effet, pas possible d'ajouter une valeur de type simple telle que int, float ou double.

6.3 La classe Vector

Un objet de classe Vector peut « grandir » automatiquement d'un certain nombre de cellules pendant l'exécution, c'est le programmeur qui peut fixer la valeur d'augmentation du nombre de cellules supplémentaires dès que la capacité maximale en cours est dépassée. Dans le cas où la valeur d'augmentation n'est pas fixée, c'est la machine virtuelle java qui procède à une augmentation par défaut (doublement dès le maximum est atteint).

Le type Vector est utilisé uniquement dans le cas d'objets et non d'éléments de type élémentaire (byte, short, int, long et char ne sont pas autorisés), comme par exemple le String ou tout autre objet de java que vous créer vous-même.

Les principales méthodes permettant de manipuler les éléments d'un Vector sont :

addElement(objet)	Ajoute un élément objet à la fin du vecteur.
elementAt(indice)	Retourne l'élément situé au rang indice donné.
contains(objet)	Retourne true si l'objet appartient au vecteur
isEmpty()	Retourne true si le vecteur est vide
clear()	Supprime tous les éléments du vecteur
indexOf(objet)	Retourne l'indice de objet donné en paramètre, ou -1 si objet n'existe pas.
remove(indice)	Supprime l'objet dont l'indice est spécifié en paramètre.
Void setElementAt(objet, i)	Remplace l'élément situé en position i par l'objet spécifié en paramètre.
Size()	Retourne le nombre d'éléments du vecteur. Le nombre d'objets qui y ont été insérés

Exemple :

```
public class Point {
	private double x,y;
	public Point(double x,double y){
		this.x=x;this.y=y;
	}
	public void setx(double x) {
		this.x=x;
	}
	public void sety(double y){
		this.y=y;
	}
	public double getx(){
		return x;
	}
	public double gety(){
		return y;
	}
	public String toString(){
		return ("("+x+","+y+")");
	}
	public void affichePoint(){
		System.out.println(toString());
	}
} // fin Point

import java.util.Scanner;
import java.util.Vector;
public class MainVector {

	public static void afficheVector(Vector V){
		Point p;
		for (int i=0;i<V.size();i++) {
			p=(Point)V.elementAt(i);
			System.out.print("V [ "+i+ " ]");
			p.affichePoint();
		}
	}
	public static void main(String[] args) {

		Scanner E=new Scanner(System.in);
		Vector tableau = new Vector();
		Point p;  int i;
		System.out.println("Donnez 4 points :");
```

```
        //  Initialisation du vecteur
                for (i=0;i<4;i++){
                        System.out.print(i + ": ");
                        p=new Point(E.nextDouble(),E.nextDouble());
                        tableau.addElement(p);
                }
                afficheVector(tableau);

                tableau.remove(2); // supprime le point se trouvant à la position 2
                afficheVector(tableau);

                p=new Point(0.5,-0.5);
                // remplace le point  se trouvant à la position 0 par le point p
                tableau.setElementAt(p,0);
                afficheVector(tableau);

                // affiche la capacité du vecteur ici =10 (taille max)
                System.out.println("Capacite : "+ tableau.capacity());
        } // fin main
} // fin MainVector
```

Les collections stockent tout type d'objets

Comme nous l'avons déjà mentionné l'ajout d'une valeur dans une liste ne peut s'effectuer qu'au travers d'objets. Il est impossible d'insérer un simple entier dans une liste comme par exemple :

```
ArrayList  listeValeur = new ArrayList() ;
    listeValeur.add(100) ;
```

Ici l'ajout de la valeur numérique 100 provoque une erreur de compilation ayant pour message « The method add (int, Object) in the type ArrayList is not applicable for the arguments (int) »

Pour insérer une valeur numérique, il convient de forcer le type simple (int, char, double etc .) à devenir un objet comme le montre l'instruction suivante :

```
listeValeur.add(new Integer(100)) ;
```

Dans cette situation, la valeur numérique 100 est transformée en un « objet » contenant la valeur 100. Cette dernière peut alors être insérée dans une liste de type ArrayList.

Cela fait, rien ne nous interdit d'insérer dans la même liste une nouvelle valeur comme suit :

```
listeValeur.add("22") ;
```

La chaîne de caractères "22" n'est pas un nombre mais un objet de type String. Il peut donc être insérer dans la liste listeValeur sans aucune erreur ne soit détectée lors de la compilation. La liste contient deux objets de type différents.
Cependant un problème se pose lors de la consultation de la liste. En effet, pour connaître la valeur enregistrée dans une liste il convient de connaître son type. Pour récupérer la valeur 100 nous devons écrire :

Integer valeur1= (Integer) listeValeur.get(0) ;

Mais il n'est pas possible de récupérer la valeur "22" de la même façon. En effet, écrire :

Integer valeur2 = (Integer) listeValeur.get(1) ;
provoque une erreur d'exécution ayant pour message « Exception in thread "main" java.lang.ClassCastException : java.lang.String ». La solution consistant à transformer la valeur "22" en String est valide. Mais elle oblige à connaître avant consultation, le type de chaque donnée enregistrée, pour chaque indice. Ce qui peut devenir très vite complexe à gérer.

Les génériques forcent le contrôle du type de données

Pour éviter ce type d'erreur, la solution consiste à forcer le type de la liste à une seule forme de données en utilisant les types génériques. Grâce à cette nouvelle structure, les listes sont déclarées comme étant une liste d'entiers ou de chaines de caractères ou encore d'étudiants. Ils ne peuvent contenir aucune autre forme de données que celles spécifiées lors de la déclaration.

La structure de déclaration d'une liste utilisant des types génériques s'écrit :

ArrayList<Integer> listeValeur = new ArrayList<Integer>() ; si la liste ne doit contenir que des entiers, ou encore :

ArrayList<Etudiant> listeValeur = new ArrayList<Etudiant>() ; pour définir une liste ne contenant que des objets de type Etudiant.

Le terme <type> permet d'indiquer au compilateur quel type de données peut être traité par la liste ainsi créée.

Les génériques simplifient le parcours des listes

Grâce aux types génériques et à la nouvelle syntaxe de la boucle for, l'affichage des éléments d'une liste est simplifié. Ainsi avant nous devions pour parcourir une liste d'étudiants, écrire les instructions suivantes :

```
Etudiant  tmp ;
  for (int i=0 ; i<nbEtudiants ;i++)  {
    tmp=(Etudiant) liste.get(i);  ← cast
    tmp.afficheUnEtudiant();
  }
```

Avec les types génériques il n'est plus besoin d'utiliser le mécanisme du cast , et la boucle for s'écrit beaucoup plus simplement comme le montre les instructions ci-après :

```
ArrayList<Etudiant> listeEtudiant = new ArrayList<Etudiant>() ;
for (Etudiant e : listeEtudiant) e.afficheUnEtudiant() ;
```

La boucle for de traduit littéralement de la façon suivante : « pour chaque étudiant ***e*** contenu dans la liste listeEtudiant, afficher son contenu ».

7. Fichiers d'objets structurés

```
public  class Main{
…
    public static void main(String[] args) {

      …   // lecture des données  et creation de l'objet TypeStr

 TypeStr E=new TypeStr(a,b,c,d);


// J'ouvre un fichier en écriture

        ObjectOutputStream os = null;
        try {   os = new ObjectOutputStream(new FileOutputStream("Tournoi"));
        } catch (FileNotFoundException e) {
                        e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }

       try {    os.writeObject(E);
        } catch (IOException e) {
                e.printStackTrace();
        }

    try {     os.close();
        } catch (IOException e) {
                e.printStackTrace();
        }

//  je rouvre le fichier en lecture

    ObjectInputStream is = null;
        try {   is = new ObjectInputStream(new FileInputStream("Tournoi"));
        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }

    try {       E=(TypeStr) is.readObject();
        } catch (IOException e) {
                e.printStackTrace();
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        }
```

```
    try {       is.close();
         } catch (IOException e) {
                e.printStackTrace();
         }
…
} // fin method main
}// fin classe Main
```

// Il faut rajouter implements Serializable

```
public class TypeStr implements Serializable {

private int a;

private double b;

private String c;

private String[] d=new String[23];

public TypeStr(int a, double b, String c, String[] d){

  …

}

…

} // fin de la classe TypeStr
```