جامعة الحسن الأول
UNIVERSITÉ HASSAN 1ER

UNIVERSITE HASSAN 1ER
ÉCOLE NATIONALE DES SCIENCES APPLIQUEES – BERRECHID

ENSA Ecole Nationale des Sciences Appliquées BERRECHID

POLYCOPIE DES TRAVAUX PRATIQUES

PROGRAMMATION OBJET JAVA

ROFESSEUR : LAHCEN MOUMOUN

DEPARTEMENT GENIE INFORMATIQUE & MATHEMATIQUES

Module : Programmation Orientée Objet (JAVA)

Chapitre 1: Eléments de base du langage JAVA

Pr LAHCEN MOUMOUN
ensablearn@gmail.com

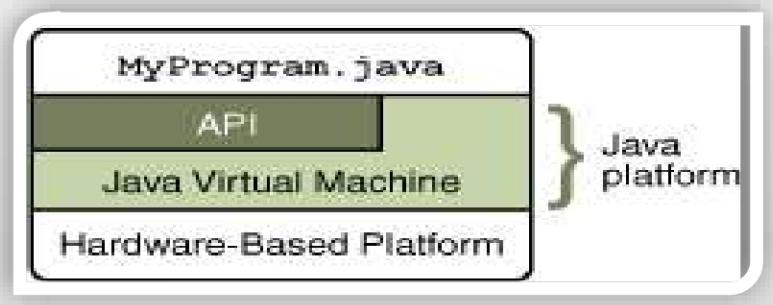
Le langage JAVA

- Java est un langage de programmation développé par Sun Microsystems.
- Ses principales caractéristiques sont:
  - C'est un langage orienté objet dérivé du C, mais plus simple et souple que le C++ (la libération de l'espace mémoire est prise en charge un gestionnaire de détection des objets à détruire appelé ramassse-miettes et en anglais «Garbage collector»).
  - Il est multi-plateforme : tous les programmes Java tourneront sans modification sur toutes les plateformes où existe Java.
  - Il est doté en standard d'une riche bibliothèque de classes, comprenant la gestion des interfaces graphiques (fenêtres, boites de dialogue, contrôles, menus, graphisme), la gestion des exceptions, les accès aux fichiers et au réseau (notamment Internet),...

La plateforme JAVA

La plateforme Java a deux composants :

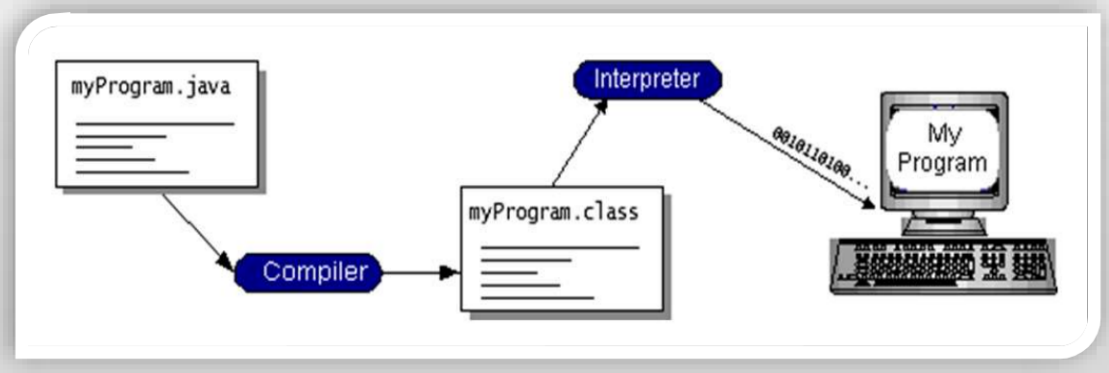
- L'API Java (Java Application Programming Interface): une collection de composants réutilisables regroupés en bibliothèques connus par Packages.
- La Machine Virtuelle Java (**JVM**: Java Virtual Machine) : programme permettant d'interpréter et d'exécuter un programme Java.



La plateforme JAVA

En Java, le code source (fichier .java) n'est pas traduit directement dans le langage de l'ordinateur :

- La compilation traduit le fichier source dans un langage intermédiaire indépendant de la plateforme appelé « bytecode » (un pseudo langage universel, disposant des fonctionnalités communes à toutes les machines).
- Le « bytecode » (fichier .class) est interprété par la Machine Virtuelle Java.



Structure élémentaire d'un programme

La structure élémentaire d'un programme JAVA est :

```
public class NomClasse {
   public static void main (String [ ] args) {  ...  }
 }
```

- La première lettre du nom de la classe doit être une lettre majuscule.
- La méthode main est une méthode de classe publique, qui contient le programme principal à exécuter.
- static : ne nécessite pas de création d'objet pour être exécuté.

Exemple

```
public class MyProgram{
   public static void main (String args[]) {
                System.out.println ("Mon premier programme Java") ;  }
}
```


L'environnement de travail

Pour exécuter un programme Java on peut utiliser :

- L'outil de base JDK (Java Development Kit) «http://www.oracle.com/technetwork/java/javase/downloads/index.html » qui comprend le compilateur, le débogueur et le générateur de documentation.
- Des environnements de développements intégrés (gratuits):
  - Eclipse : eclipse.org/
  - NetBeans : netbeans.org/
  - …

Java est en évolution -permanente : mise à jour de la machine virtuelle et ajout de nouvelle API.

Instructions de base

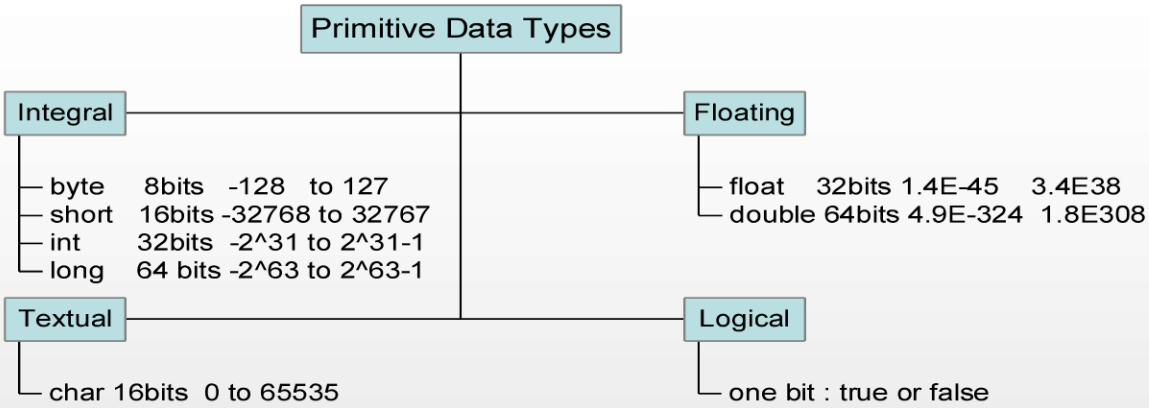
- Comme les autres langages, Java distingue les instructions de déclaration et les instructions exécutables.
- Les instructions exécutables, peuvent être classer en trois catégories :
  - Les instructions simples, obligatoirement terminées par un point-virgule.
  - Les instructions de structuration telles que if ou for
  - Des blocs, délimités par { et }.
- On dispose de deux formes de commentaires :
  - Les commentaires usuels, placés entre « /* » et « */ » (utiliser la syntaxe « /** commentaire */ » pour la documentation « javadoc »)
  - Les commentaires de fin de ligne placés après « // ».

Les opérateurs

- Opérateur d'affectation : **=** (L'opérateur permet d'enchaîner plusieurs affectations, exemple : i=j=k=2;)
- Opérateurs arithmétiques : ***** , **/** , **%** (reste de la division entière) , **+** , –
- Les opérateurs relationnels : **<** , **>** , **<=** , **>=** , **==**(test d'égalité), **!=** (test d'inégalité)
- Les opérateurs logiques : **!** (non logique), **&&** et **&** (et logique) , **| |** et **|** (ou inclusif), **^** (ou exclusif). & et | évaluent toujours leurs deux opérandes. Avec && et ||, le second opérande peut ne pas être évalué.
- Opérateur d'abréviation : += , -= , *= et /=
  « variable **+=** expression » est similaire à « variable**=** variable **+** expression »
- Opérateur d'incrémentations et de décrémentation : ++ et -- (variable ++ ≡ variable = variable +1 , variable -- ≡ variable = variable – 1)

```
i =5;
n = ++i – 5 // i=6 et n=1
n = i++ – 5 // i=6 et n=0  ici la valeur de l'expression i++ est 5
```

Types primitifs



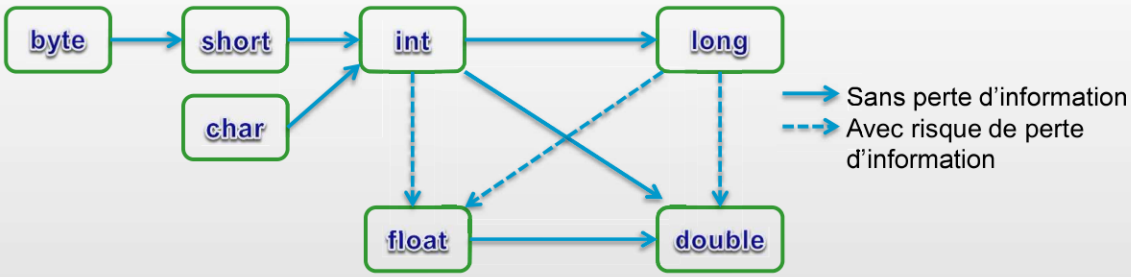
Exemple de déclaration :

```
int    x;
long x= 200L; // Nombre accolé à un L
float x= 23.233F; // Nombre accolé à un F
char c='a'; char c='\u0061'; // Notation hexadécimale d'un caractère Unicode
```

- Une variable doit être initialisée avant d'être utilisée.
- Le mot-clé final (utilisé pour définir une constante) permet de bloquer la modification de la valeur d'une variable (exemple: final double PI=3.14d;).

Conversions implicite de types

Conversion d'ajustement de type: une opération sur deux valeurs de type identique fournie un résultat de même type (exemple : 5/2 vaut 2). Dans le cas où ces types sont différents , une conversion d'ajustement est effectuée (exemple 5.0/2 vaut approximativement 2.5) selon le schéma suivant:



Exemple:

```
int nb ;short p ; byte b1=10, b2 =108;
n = b1 * b2 ; // OK car l'expression b1 * b2 est de type int
p = b1 * b2 ; // erreur de compilation : on ne peut affecter int a short
…
int n = 123456789;
float f = n; // f vaut 1.23456792E8
```

Conversions explicite de types

Appelée aussi opération de **cast** (ou Transtypages) consiste en une modification de type de donnée forcée (utilisation de type de donnée comme opérateur de cast pour spécifier la conversion)

Exemple :

```
int n=10, p= 3; double d1,d2;
d1=n/p; d2 =(double) n/p // d1=3 et d2=3.3333…
int n = 1000 ;byte b ;
b = (byte) n ; // légal conserver que les 8 bits les moins significatifs b=-24
float x ; double y ; int n ; short p ;
y = 1e+300 ; x = (float) y ;  //  x= Infinity  càd valeurs infinies
x = 123456789.f ; p = (short) x ; // conversion en int  puis en short
                                  //x=1.23456792E8 p= -13032
```


Portée des variables

- Les variables peuvent être déclarées n'importe où dans un bloc.
- Les variables globales n'existent pas.

```
…
{
    int x;
     …
    {
      int y;
       …
    }
    …
}
…
```

porté de y

porté de x

Une seconde variable x dans le second bloc ne peut pas être déclarée.

Les structures de contrôle: l'instruction if

L'instruction de test « if » permet de choisir l'instruction ou le bloc d'instructions à exécuter selon l'état d'une condition (valeur vraie ou fausse)

Syntaxe : **if** (condition) bloc_vrai [**else** bloc_faux]

- La condition doit toujours être entourée par des parenthèses
- Une alternative (bloc vrai ou bloc faux) qui comprend plusieurs instructions doit être délimitée par des accolades
- La partie fausse est facultative (les crochets [...] signifient que ce qu'ils renferment est facultatif)
- Dans le cas d'une condition imbriquée, Un else se rapporte toujours au dernier if rencontré auquel un else n'a pas encore été attribué.
- Au lieu d'écrire : if i>j z=a ; else z= b ;

  On peut écrire : z=i>j ? a : b ;

Les instructions de branchement

- L'instruction « **break** » permet d'interrompre le déroulement d'une boucle, en passant à l'instruction suivant la boucle. En cas de boucles imbriquées, l'instruction « **break** » fait sortir de la boucle la plus interne. Cependant, Java permet de faire suivre ce mot-clé « **break** » d'une étiquette qui doit alors figurer devant la structure dont on souhaite sortir.
- L'instruction « **continue** » permet de passer prématurément au tour de boucle suivant. En cas de boucles imbriquées, l'instruction « **continue** » ne concerne que la boucle la plus interne. Comme pour « **break** », Java permet de faire suivre ce mot-clé « **continue** » d'une étiquette qui doit alors figurer devant la boucle sur laquelle on souhaite boucler.

Les structures de contrôle: L'instruction switch

L'instruction switch effectue un aiguillage vers une ou plusieurs instructions en fonction du contenu d'une variable contrôle.

Syntaxe:

```
switch (expression){
    case valeur1 : Bloc1;  break ;
    …
    case valeurn : Blocn ; break ;
    default : Blocf
}
```

- Les valeurs de l'expression doivent être des types byte, short, char ou int (valeur entière ou caractère)
- L'instruction break permet de sortir directement du bloc de switch.

Les structures de contrôle: les boucles do et while

- L'instruction «do» et « while » permettent de répéter l'exécution d'un bloc d'instructions tant qu'une une condition est vraie (on ne sait pas a priori le nombre de répétitions à faire)

Syntaxe:

```
while (condition ) bloc_instructions
do bloc_instruction  while (condition ) ;
```

- Avec l'instruction « while », la condition est évaluée avant chaque parcours de la boucle contrairement à l'instruction « do » (la boucle « while » peut très bien n'être parcourue aucune fois si la condition est fausse dés qu'on l'aborde alors que la boucle do est toujours parcourue au mois une fois).

Les structures de contrôle: la boucle for

- L'instruction «for» est généralement utilisée pour répéter un bloc un nombre de fois prédéterminé.

  Syntaxe: **for** (expinit **;** condition **;** expfinale) bloc_instruction

  - expinit : Instruction de démarrage (en cas de plusieurs initialisions, les séparées par des virgules)
  - condition : Les instructions de la boucle sont exécutées tant que cette condition est vérifiée (évaluation de la condition avant la prochaine exécution des instructions de la boucle)
  - expfinale : Exécutée à chaque fin de la boule (avant un nouveau parcours de la boucle) ; en cas plusieurs instructions on doit les séparées par des virgules
- A partir de la version 5 du JDK, une nouvelle structure de boucle souvent nommée for... Each a été introduite. Elle ne s'applique toutefois qu'au parcours des éléments d'une collection ou d'un tableau.

Exercice 1

Les nombres de Amstrong appelés parfois nombres cubes sont des nombres entiers qui ont la particularité d'être égaux à la somme des cubes de leurs chiffres. Par exemple, 153 est un nombre de Amstrong car on a : $153 = 1^3 + 5^3 + 3^3$.

Afficher en java tous les nombres de Amstrong sachant qu'ils sont tous compris entre 100 et 499. Indication : Les nombres de Amstrong sont : 153, 370, 371 et 407.

Exercice 2

Dans cet exercice nous rappelons les notions suivantes :

- On appelle nombre premier tout entier naturel supérieur à 1 qui possède exactement deux diviseurs, lui-même et l'unité ;
- On appelle diviseur propre de n, un diviseur quelconque de n, n exclu ;
- Un entier naturel est dit parfait s'il est égal à la somme de tous ses diviseurs propres ;
- Les nombres n tels que : $(n + a + a^2)$ est premier pour tout a tel que $0 \leq a < (an - 1)$, sont appelés nombres chanceux.

Écrire le code permettant de définir les quatre fonctions : somDiv, estParfait, estPremier et estChanceux :

- La fonction somDiv retourne la somme des diviseurs propres de son argument ;
- Les trois autres fonctions vérifient la propriété donnée par leur définition et retourne un booléen. Plus précisément, si par exemple la fonction estPremier vérifie que son argument est premier, elle retourne True, sinon elle retourne False.

Ecrire un programme principal qui comporte une boucle de parcours de l'intervalle [2, 1000] incluant les tests nécessaires pour afficher les nombres parfaits et chanceux.

Université Hassan 1er
École Nationale des Sciences Appliquées – Berrechid

Module : Programmation Orientée Objet (JAVA)

Chapitre 2: Les classes et les objets en JAVA

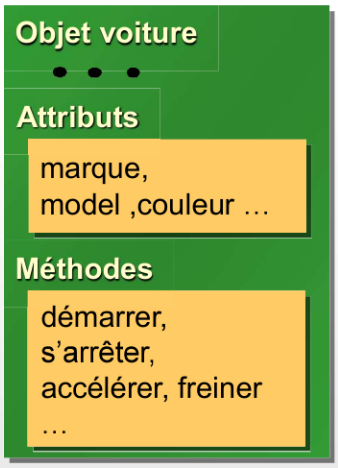
Pr LAHCEN MOUMOUN
ensablearn@gmail.com

Paradigme orienté objet

La programmation orientée objet (POO), est une façon d'architecturer une application informatique en regroupant les caractéristiques d'un objet , appelées attributs et ses comportements, appelés méthodes au sein d'une même entité qu'on appelle un OBJET.

Notion d'attribut et méthode

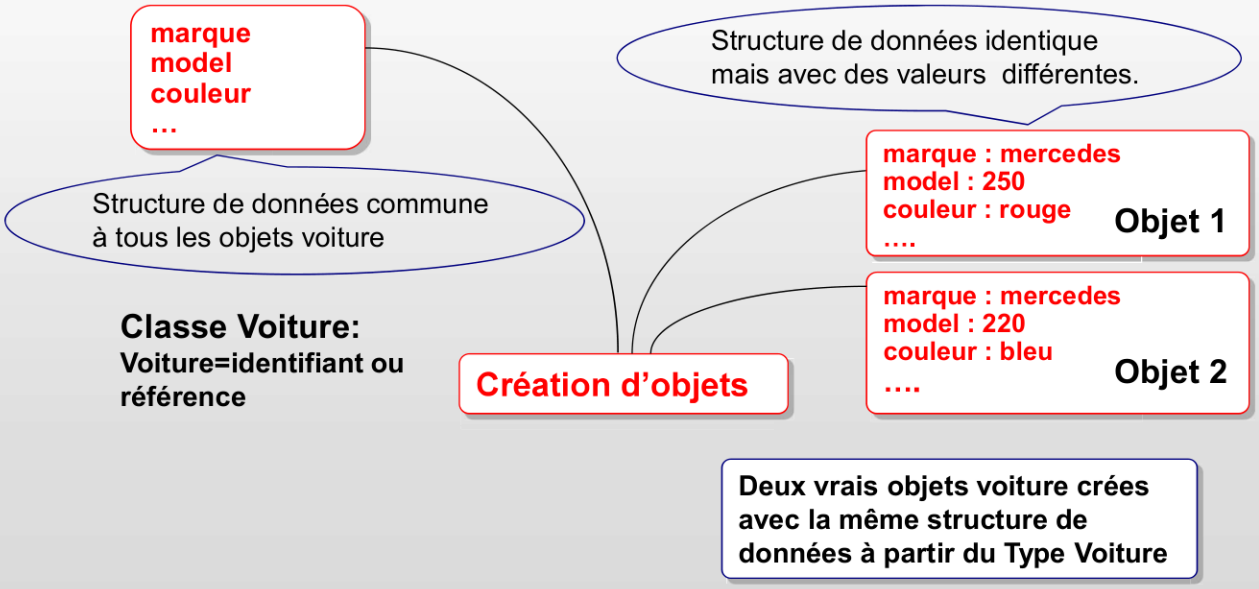
- Un attribut est une propriété de l'objet, il est propre et particulier à un être, à quelqu'un ou à quelque chose.
- Une méthode définit un comportement particulier de l'objet. C'est une tâche que l'objet exécute lorsqu'un autre objet la lui demande.



- La signature d'une méthode est composée du nom de la méthode ainsi que de la liste des identifiants et types des arguments de la méthode.
- L'état d'un objet est défini par les valeurs de ses attributs. L'état de l'objet n'est pas fixe dans le temps, il change à chaque fois qu'on modifie la valeur d'un attribut de l'objet.

Notion de classe

Une classe représente un modèle de construction d'un objet, Il s'agit d'une description abstraite en termes de données (attributs) et de comportements(méthodes) d'une famille d'objets.



Représentation graphique des classes et objets

Classe

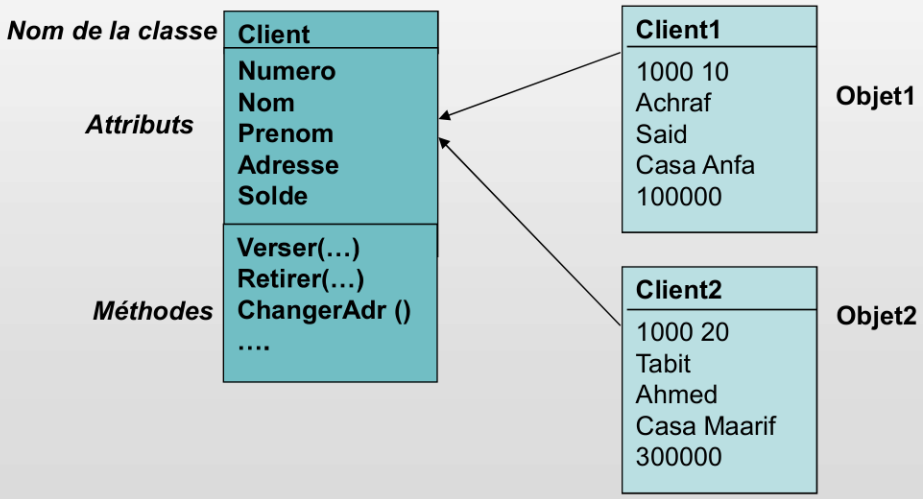
Objets

Voiture1 :Voiture
Couleur : rouge
numSerie : 54FD
vitesse : 0
vitesseMax : 100
moteur : true
plaque : ABCD
nbrePlaces : 6

Voiture2 :Voiture
Couleur : bleu
numSerie : 3321
vitesse : 0
vitesseMax : 160
moteur : true
plaque : GRTD
nbrePlaces : 6

Exemple d'objet et de classe

Le client d'une banque possède un numéro (numéro de compte), un nom, un prénom, une adresse, un solde,… elle peut verser et retirer de l'argent de son compte, changer son adresse,…



Classe et instanciation d'objets

Moule représentant la classe « Gâteau »

Les objets gâteau crées à partir du moule

- La création d'un objet à partir d'une classe s'appelle: l'INSTANCIATION
- Le ''point'' permet d'accéder à tous ce que contient l'objet, que ce soit les attributs ou les méthodes.

```
voiture1.marque = "Mercedes »
voiture1. démarrer()
```

Structure générale d'une classe JAVA

- Syntaxe de déclaration :

```
[Mode d'accès ] class IdentificateurClasse  {
    [ Mode d'accès ] nom_type identificateur_attribut;
    …
    [Mode d'accès] type_retour nom_methode([liste_parametres formels]) {
        // instructions...
    }
```

Mode d'accès	Classe	Attribut | méthode
public	Visible en dehors de son package	Accessible partout où la classe est accessible.
private	Visible uniquement par les classes du fichier source	Visible uniquement par les méthodes de la classe
Aucun	Visible uniquement dans le package de la classe	Accessible dans tout le paquetage où la classe est définie

- Syntaxe d'instanciation :

```
IdentificateurClasse IdentificateurReference= new  IdentificateurClasse(…);
```


Instanciation des objets

- Dès qu'un objet est créé ses attributs sont initialisés à une valeur par défaut ainsi définie :

Type du champ	Valeur par défaut
boolean	false
char	caractère de code nul
entier (byte, short, int, long)	0
flottant (float, double)	0.f ou 0.d

- Un attribut d'une classe peut très bien être la référence à un objet. Dans ce cas, cette référence est initialisée à une valeur conventionnelle notée null.
- Il est possible de bloquer la modification de la valeur d'un attribut (en dehors de l'instanciation) à l'aide du mot-clé final.

Exemple de classe

Classe Point

```
public class Point {
   public void initialise (int abs, int ord) {  x = abs ; y = ord ; }
   public void deplace (int dx, int dy) { x += dx ; y += dy ;  }
   public void affiche () {
       System.out.println ("Je suis un point de coordonnees " + x + " " + y) ;  }
   private int x ; // abscisse
   private int y ; // ordonnee
}
```

Programme(classe) de test de la classe Point

```
public class TestPoint1{
    public static void main (String args[]) {
       Point a ;  a = new Point() ;
       a.initialise(3, 5) ; a.affiche() ; /* l'instruction x=3; est interdite ( accès réservé
                                              aux méthodes de la classe Point )  */
       a.deplace(2, 0) ; a.affiche() ;
       Point b = new Point()  ; b.affiche() ; b.initialise (6, 8) ; b.affiche() ;
    }
}
```


Les méthodes d'accès et d'altération d'une classe

- Les méthodes d'accès (en anglais accessor, appelé souvent getter) fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses champs (généralement privés), sans les modifier ;
- Les méthodes d'altération (en anglais mutator, appelé souvent setter) qui modifient l'état d'un objet, donc les valeurs de certains de ses champs.

Exemple

```
public class Point {
   …
   public int getX() { return x ; }
   public int getY() { return y ; }
   public void setX (int abs){x=abs;}
   public void setY (int ord){y=ord;}
   public void setPosition(int abs, int ord)
          { x = abs ; y = ord ; }
   public Point copie(){
          Point p = new Point() ; p.x=x; p.y=y; return (p);}
   … }
```

```
public class TestPoint2 {
  public static void main (String args[]) {
     Point a  = new Point(); …
     System.out.print  ("X="+a.getX());
     System.out.println (" Y="+a.getY());
     Point b=a; // références même objet
     Point c=a.copie(); //clonage (2objets)
     … }
}
```

Surcharge des méthodes

- Dans une même classe, il est possible de définir plusieurs méthodes qui portent le même nom (surcharge ou surdéfinition), le compilateur peut les différencier en regardant la liste de type des paramètres formels de chaque méthodes (signatures des méthodes).
- À la rencontre d'un appel de méthode, le compilateur recherche toutes les méthodes acceptables et il choisit la meilleure (si elle existe). Pour qu'une méthode soit acceptable, il faut :
  - Qu'elle dispose du nombre d'arguments effectifs voulus.
  - Que le type de chaque argument effectif d'appel soit compatible par affectation avec le type du paramètre correspondant.
- Le type de la valeur de retour d'une méthode surdéfinie n'intervient pas dans le choix de la méthode acceptable.

Exemple de surcharge des méthodes

Classe Point

```
public class Point {
  …
  public void deplace (int dx, int dy) {
        x += dx ; y += dy ; }
  public void deplace (int dxy) {
        x += dxy ;  y += dxy ; }
  …
}
```

Classe TestPoint3

```
public class TestPoint3{
   public static void main (String args[]) {
      Point a = new Point() ;
      a.initialise(10, 5) ; a.affiche() ;
      a.deplace(2, 0) ; a.affiche() ;
      a.deplace(3) ; a.affiche() ;
   }
}
```

La notion de constructeur

- Un constructeur est une méthode qui permet d'automatiser le mécanisme d'initialisation et l'exécution des actions utiles à la construction et au bon fonctionnement de l'objet.
  - Un constructeur porte le même nom que sa classe. Il peut être surdéfini. Il ne renvoie aucune valeur (aucun type ne doit figurer devant son nom).
  - Un constructeur ne peut pas être appelé directement depuis une autre méthode. Cependant, Un constructeur peut appeler un autre constructeur de la même classe.
  - Le corps du constructeur est toujours exécuté qu'après l'initialisation par défaut et l'initialisation explicite (initialisation figurant dans la déclaration) des attributs.
- Les méthodes d'accès (en anglais accessor, appelé souvent getter) fournissent des informations relatives à l'état d'un objet, c'est-à-dire aux valeurs de certains de ses champs (généralement privés), sans les modifier;
- Les méthodes d'altération (en anglais mutator, appelé souvent setter) qui modifient l'état d'un objet, donc les valeurs de certains de ses champs.

Exemple de constructeur

```
public class Point {
    public Point (int abs, int ord)  { // constructeur
            x = abs ; y = ord ;
    }
    public void deplace (int dx, int dy) {  x += dx ; y += dy ; }
    public void affiche ()  {
            System.out.println ("Je suis un point de coordonnees " + x + " " + y) ; }
    private int x ;
    private int y ;
}
public class TestPoint4{
    public static void main (String args[]) {
            Point a  = new Point(3, 5) ; // appel automatique du constructeur
            a.affiche() ; a.deplace(2, 0) ; a.affiche() ;
            Point b = new Point(6, 8) ; // appel automatique du constructeur
            b.affiche() ;
    }
}
```

Exercice

Quels résultats fournit le programme suivant ?

```
class A {
  public A (int dec, int coeff) {
    decal=dec;
    nbre *= coeff ; nbre += decal ; }
  public void affiche () {
    System.out.println ("nbre = " + nbre + " decal = " + decal) ; }
  private int nbre = 20 ;
  private int decal ;
}
public class Exercice1Ch3{
  public static void main(String[] args) {
    A a = new A (10, 5) ; a.affiche() ;
  }
}
```


Exemple

```
public class Point {
   public Point () { }
   public Point (int abs, int ord)  {
       x = abs ; y = ord ;}
   public Point (int x){
            this.x = this.y = x; }
    public Point (Point a) {
            this(a.x, a.y);}
   public int getX() { return x ; }
    public int getY() { return y ; }
    public void setX (int abs){x=abs;}
    public void setY (int ord){y=ord;}
    public void setPosition(int abs, int ord){
        x = abs ; y = ord ; }
    public void deplace (int dx, int dy) {
         x += dx ; y += dy ; }
     public String toString()  {
            return ("Point [ " + x + ", " + y + "]" ) ; }
     private int x ;
     private int y ;
}
```

```
public class TestPoint3 {
  private static void affiche (String st) {
        System.out.print(st) ; }
  public static void main (String args[]) {
     Point a  = new Point() ;
     Point b = new Point( 8) ;
     affiche(b.toString()) ;
     a.setX(b.getX()+1);a.setY(b.getY());
     affiche(a.toString()) ;
     Point c=a; Point d=new Point(c);
      b=a;
     affiche(b.toString()) ;}
}
```

- Le mot-clé **this** désigne l'instance courante de la classe.
- L'appel **this**(...) permet d'appeller un autre constructeur de la même classe (1ière instruction obligatoirement).
- mot-clé **static** indique que l'exécution ne nécessite pas la création d'objet.

Le ramasse-miettes

La destruction des objets Java est basée sur un mécanisme de gestion automatique de la mémoire nommé ramassse-miettes (en anglais Garbage Collector). Son principe est le suivant :

- Le mécanisme détermine (à des fréquences de temps données) le nombre de références à un objet donné (un objet est toujours géré par référence).
- Lorsqu'il n'existe plus aucune référence sur un objet (objet inaccessible). Cet objet est candidat afin de libérer l'emplacement correspondant.

Attributs et méthodes statiques et objets membres

- Les méthodes statiques peuvent fournir des services n'ayant de signification que pour la classe même. Elles sont définies pour être appelée indépendamment de tout objet de la classe.
- Les méthodes statiques sont aussi utilisées pour regrouper au sein d'une classe des fonctionnalités ayant un point commun et n'étant pas liées à un quelconque objet (par exemple la classe Math qui contient des fonctions de classe telles que sqrt, sin, cos). Ce regroupement est le seul moyen dont on dispose en Java pour retrouver (artificiellement) la notion de fonction indépendante.
- Un attribut statique est une donnée définie et partagée par toutes les instances d'une même classe.

Exemple d'attributs et méthodes statiques

```
public class Point {
    …
    public static boolean coincide (Point p1, Point p2) {
        return ((p1.x == p2.x) && (p1.y == p2.y)) ;}
    …
    private static long nbPoints=0l;
}
public class TestPoint5{
    public static void main(String[] args) {
        Point a  = new Point();
        System.out.println  ("nb de points: "+Point.getNbpoints());
        Point b=a;  System.out.println  ("nb de points: "+Point.getNbpoints());
        Point c;
        System.out.println  ("nb de points: "+Point.getNbpoints());
        c=new Point();
        System.out.println  ("nb de points: "+Point.getNbpoints());
        System.out.println  (Point.coincide(a,c));
    }
}
```


Les objets membres

- Les attributs d'une classe sont soit d'un type primitif, soit des références à des objets. Ces objets sont souvent appelés objets membres ou encore objets composants. Cette relation de de type contenant-contenu peut être qualifiée soit d'agrégation ou de composition.
- L' Agrégation entre 2 classes est une relation d'un élément dans un ensemble. Cependant, la relation de composition est une forme d'agrégation avec un couplage encore plus fort : la destruction de l'agrégat (l'objet en composite) entraîne la destruction des composants agrégés (les objets composants).

Exemple de Les objets membres

```
public class Point {
    public Point(int x, int y) {
            this.x = x ;  this.y = y ; nbPoints++; }
    …
    public static boolean coincide (Point p1, Point p2) {
            return ((p1.x == p2.x) && (p1.y == p2.y)) ;}
    public static long getNbPoints(){ return (nbPoints);}
    private int x, y ;
    private static long nbPoints=0l;
}
public class Cercle {
    public Cercle (int x, int y, float r) {  c = new Point (x, y) ;  this.r = r ;  }
     public Cercle (Point p , float r) {  c =p ;  this.r = r ;  }
     …
    public String toString() {
            return ("Cercle de rayon " + r  + " et de centre " + c.toString()) ;   }
    public void deplace (int dx, int dy) {  c.setX (c.getX() + dx); c.setY (c.getY() + dy) ; }
    private Point c ; // centre du cercle
    private float r ;   // rayon du cercle
}
```

Une classe Cercle permet de représenter des cercles définis par un centre (objet du type Point) et un rayon (flottant).

Les packages

- La notion de package correspond à un regroupement logique sous un identificateur commun d'un ensemble de classes.
- L'attribution d'un package se fait au début du fichier source par l'instruction : « **package *NomPackage*** ; » .
- Pour utiliser une classe appartenant à un package (différent du package par défaut), on peut :
  - Ajouter le nom complet du package devant chaque nom de classe (par exemple java.util.Date today = new java.util.Date();)
  - Utiliser la directive **import** pour faire référence soit à une classe particulière d'un package , soit tout un paquetage (par exemple : import Utilitaires; import Utilitaires.*; import Utilitaires.Tris;)
- il existe un paquetage particulier nommé java.lang (classe Math, System, Float ou Integer qui est automatiquement importé par le compilateur sans avoir à introduire d'instruction import.

La classe String

- Les chaînes de caractères sont des instances de la classe String (paquetage java.lang).
- La classe String dispose de deux constructeurs, l'un sans argument créant une chaîne vide, l'autre avec un argument de type String
  - **String** ch1 = new **String** () ; // ch1 contient la référence à une chaine vide
  - **String** ch2 = new **String**("JAVA") ; // ch2 contient la référence à "JAVA"
  - **String** ch3 = "JAVA"; // ch3 contient la référence à la chaine à "JAVA"
- L'opérateur «+» est défini lorsque ses deux opérandes sont des chaînes. Il fournit en résultat une nouvelle chaîne formée de la concaténation des deux autres.
- Comme dans le contexte numérique, l'opérateur « +=» s'applique également lorsque son premier opérande est de type chaîne. Il fournie , ainsi, un résultats de type String.

```
String ch = "chiffres = " ;
for (int i = 0 ; i<=9 ; i++)  ch += i ;
System.out.println (ch) ; // affiche : chiffres = 0123456789
```


La classe String

- La classe String offre de nombreuses méthodes (un objet String n'est pas modifiable):
  - **length( )** : Renvoie la longueur de la chaîne de caractères.
  - **charAt(int index)** : Retourne le caractère de la chaine dont le rang est index (le premier caractère porte le rang 0).
  - **equals (String str)** : compare le contenu de la chaine de caractère avec la chaine str. La méthode equalsIgnoreCase effectue la même comparaison, mais sans distinguer les majuscules des minuscules.
  - **compareTo(String str)**: Effectue une comparaison lexicographique de chaînes pour savoir laquelle de deux chaînes apparaît avant une autre, en se fondant sur l'ordre des caractères. Elle fournit un entier négatif si la chaîne arrive avant la chaîne argument str ou un entier nul si la chaîne est égale à str et un entier positif si la chaîne arrive après str.
  - **toUpperCase()** et **toLowerCase()** : permettent, respectivement, de mettre la chaîne de caractères en lettres majuscules et minuscules.
  - **concat(String str)** : ajoute l'argument str à la chaine et renvoie la nouvelle chaine.
  - **trim()** : enlève les éventuels séparateurs de début et de fin (espace, tabulations, fin de ligne).

La classe prédéfinie String

- La classe String offre de nombreuses méthodes:
  - **indexOf (char ch)** et **indexOf (String str)** : Renvoie position l'indice de la première occurrence du caractère ch ou du début de la chaîne str (retourne la valeur -1 si aucune correspondance n'a été trouvée). La méthode surdéfinie lastIndexOf permet d'effectuer les mêmes recherche que indexOf, mais en examinant la chaîne depuis sa fin.
  - **replace(char ch1, char ch2)** : renvoie la chaine dont les occurrences du caractère ch1 sont remplacées par le caractère ch2.
  - **substring(int beginIndex) , substring(int beginIndex, int endIndex) :** retourne la sous-chaîne constitué des caractères dont l'indice est compris entre beginIndex et endIndex - 1.
  - **endsWith(String str)** : vérifie si la chaine se termine par l'argument str
  - **startsWith(String ch)** : Vérifie si la chaine commence par la sous chaine str
  - …

Les tableaux en Java

- En Java les tableaux sont considérés comme des objets (l'identificateur d'un tableau est une référence).

  Syntaxe de déclaration :

  typeBase identificateurTableau[ ]; // utiliser [ ]..[ ] pour plusieurs dimensions;

  Création et allocation de l'emplacement d'un tableau :

  identificateurTableau= new typeBase[tailleTableau] ;
- L'accès à un élément d'un tableau peut se faire au moyen d'un indice. Cet indice correspond à la position de l'élément au sein du tableau (le premier indice d'un tableau est 0).
- **length** permet de connaître le nombre d'éléments d'un tableau de référence donnée: identificateurTableau.**length**
- La structure **for... each** est adaptée à la consultations de valeurs d'un tableau.

Exercice

Réaliser une classe Répertoire permettant de gérer un répertoire téléphonique associant un numéro de téléphone (chaîne de caractères) à un nom. Pour faciliter les choses, on prévoira une classe Abonne destinée à représenter un abonné et disposant des fonctionnalités indispensables.

La classe Repertoire devra disposer des fonctionnalités suivantes :

- Constructeur recevant un argument de type entier précisant le nombre maximum d'abonnés que pourra contenir le répertoire (cette particularité évite d'avoir à se soucier d'une gestion dynamique du répertoire),
- Méthode addAbonne permettant d'ajouter un nouvel abonné ; elle renverra la valeur false si le répertoire est plein, la valeur true sinon,
- Méthode getNumero fournissant le numéro associé à un nom d'abonné fourni en argument,
- Méthode getNAbonnes qui fournit le nombre d'abonnés figurant dans le répertoire,
- Méthode getAbonne fournissant l'abonné dont le rang est fourni en argument,
- Méthode getAbonnesTries fournissant un tableau des références des différents abonnés, rangés par ordre alphabétique (pour simplifier, on supposera que les noms sont écrits en minuscules, sans caractères accentués).

Écrire un petit programme de test.

Université Hassan 1er
École Nationale des Sciences Appliquées – Berrechid

Module : Programmation Orientée Objet (JAVA)

Chapitre 3: L'héritage et le polymorphisme en JAVA

Pr LAHCEN MOUMOUN
ensablearn@gmail.com

Concept d'héritage

- Le concept d'**héritage** (appelée aussi dérivation) **est synonyme de spécialisation**: un objet de la classe dérivée (ou classe fille ou la sous-classe) est considéré comme un cas particulier d'un objet de la classe de base (ou superclasse ou classe mère).
- Une classe dérivée bénéficie (hérite) des fonctionnalités de la classe de base (attributs et méthodes).

- **Syntaxe de déclaration :**

```
[public] class NomClasseDerive extends NomClasseBase
  {
    // Définition des attributs et des méthodes de la classe dérivée
    ….
  }
```


Concept d'héritage

- Le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet. L'appel du constructeur de la classe de base est, ainsi, désigné par le mot-clé super (doit obligatoirement être la 1ière instruction et ne concerne que le constructeur de la classe du niveau immédiatement supérieur).
- Un objet d'une classe dérivée accède aux membres **non privés** de sa classe de base, exactement comme s'ils étaient définis dans la classe dérivée elle-même.

Exemple

*Un objet de type Pointcol **est un cas particulier** d'un objet de type Point pour lequel une couleur à été rajoutée .*

```
class Point {    // classe de base
      public Point (int x, int y) { this.x = x ; this.y = y ; }
      …
      private int x, y ;   }
class Pointcol extends Point  {  // classe derivee de Point
      public Pointcol (int x, int y, byte couleur) {
          super (x, y) ; // obligatoirement comme premiere instruction
          this.couleur = couleur ;
      }
      …
      public String toString() {
            return (" Pointcol [" + super.toString()  + " , " + couleur + " ] ") ;   }
public class TstPcol1 {
      private static void affiche (String st) {  System.out.print(st) ; }
      public static void main (String args[ ]) {
          Point pt=new Point(2,2);    affiche(pt.toString());
          Pointcol pc2 = new Pointcol(3,5, (byte)3);  affiche(pc1.toString());
          Pointcol pc1 = new Pointcol(pt, (byte)3) ; affiche(pc2.toString());
      }
}
```


Exercice

Quels résultats fournit ce programme ?

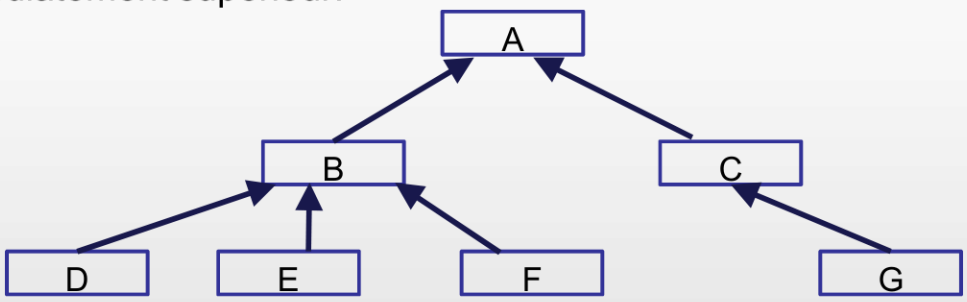
```
class A {
      public A (int nn) {
          System.out.println ("Entree Constr A - n=" + n + " p=" + p) ;
          n = nn ;  System.out.println ("Sortie Constr A - n=" + n + " p=" + p) ;}
      private int n ; // ici, exceptionnellement, pas d'encapsulation
      public int p=10 ;
}
class B extends A {
      public B (int nn, int pp) {
          super (nn) ;
          System.out.println ("Entree Constr B - n=" + n + " p=" + p + " q=" + q) ;
          p = pp ; q = 2*n ;
          System.out.println ("Sortie Constr B - n=" + n + " p=" + p + " q=" + q) ;
      }
      public int q=25 ;
}
public class TstInit {
      public static void main (String args[])
          A a = new A(5) ;   B b = new B(7, 3) ;    }
}
```

Mot clé finales et protected

- Le mot-clé **final** peut s'appliquer à une méthode ou à une classe:
  - Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.
  - Une classe déclarée final ne peut plus être dérivée (une classe finale ne pourra pas se voir ajouter de nouvelles fonctionnalités),
- Le droit d'accès dit protégé (mot-clé protected) permet de déclarer un membre accessible à des classes dérivées (qu'elles appartiennent ou non au même paquetage).

Concept d'héritage

- Dans une hiérarchie d'héritage, l'appel par super dans un constructeur ne concerne que le constructeur de la classe de base du niveau immédiatement supérieur.



- Une classe dérivée peut (en plus de la surdéfinition) redéfinir une méthode d'une classe ascendante (avec une même signature). La méthode redéfinie est appelée sur tout objet de la classe dérivée en masquant, ainsi, celle de la classe de base (la recherche d'une méthode acceptable ne se fait qu'en remontant la hiérarchie d'héritage, jamais en la descendant).

Redéfinition et surdéfinition

La surdéfinition et la redéfinition peuvent cohabiter (à éviter généralement en soignant la conception des classes).

Exemple

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

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


Le polymorphisme

Le polymorphisme se traduit par la capacité d'un objet à posséder plusieurs formes mais également la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours.

Exemple:

```
class Point { …}
class Pointcol extends Point { ...}
public class TestPoly1 {
  private static void affiche (String st) {  System.out.print(st) ; }
  public static void main (String args[ ]) {
     Point p = new Point (3, 5) ;
     Pointcol pc = new Pointcol (4, 8, (byte)2) ;
     p = pc ;          // p de type Point, reference un objet de type Pointcol
     affiche(p.toString()); // on appelle toString de Pointcol
     Point [ ] tabPts = new Point [4] ;
     tabPts [0] = new Point (0, 2) ; tabPts [1] = new Pointcol (1, 5, (byte)3) ;
     tabPts [2] = new Pointcol (2, 8, (byte)9) ; tabPts [3] = new Point (1, 2) ;
     for (int i=0 ; i< tabPts.length ; i++) affiche(tabPts[i].toString() ) ;
  }
}
```

Exemple

```
class Point {
    public Point (int x, int y) { this.x = x ; this.y = y ; }
    public void affiche () {
        identifie() ; System.out.println (" Mes coordonnees sont : " + x + " " + y) ; }
    public void identifie () {  System.out.println ("Je suis un point ") ;      }
    private int x, y ;  }
class Pointcol extends Point {
    public Pointcol (int x, int y, byte couleur) {  super (x, y) ; this.couleur = couleur ; }
    public void identifie () {
        System.out.println ("Je suis un point colore de couleur  " + couleur) ; }
    private byte couleur ;   }
public class TestPoly1 {
    public static void main (String args[ ]) {
         Point [ ] tabPts = new Point [4] ;
         tabPts [0] = new Point (0, 2) ; tabPts [1] = new Pointcol (1, 5, (byte)3) ;
         tabPts [2] = new Pointcol (2, 8, (byte)9) ; tabPts [3] = new Point (1, 2) ;
         for (int i=0 ; i< tabPts.length ; i++) tabPts[i].affiche() ;   }
}
```

*La redéfinition de la méthode **identifie** s'applique à n'importe quel objet d'une classe descendant de Point (le code de affiche de Point n'a pas besoin de connaître ce que sont ou seront les descendants de la classe).*

Exercice

Quels résultats fournit le programme suivant ?

```
class A { public void affiche() { System.out.print ("Je suis un A ") ; }   }
class B extends A {}
class C extends A{ public void affiche(){ System.out.print ("Je suis un C ");} }
class D extends C{ public void affiche(){ System.out.print ("Je suis un D "); } }
class E extends B {}
class F extends C {}
public class Poly  {
  public static void main (String arg[ ]){
    A a = new A() ; a.affiche() ; System.out.println() ;
    B b = new B() ; b.affiche() ; a = b ; a.affiche() ; System.out.println() ;
    C c = new C() ; c.affiche() ; a = c ; a.affiche() ; System.out.println() ;
    D d= new D(); d.affiche(); a=d; a.affiche(); c=d; c.affiche(); System.out.println();
    E e = new E() ; e.affiche() ;  a = e ; a.affiche() ;
    b = e ; b.affiche() ; System.out.println() ;
    F f = new F() ; f.affiche() ;  a = f ; a.affiche() ; c = f ; c.affiche() ;
  }
}
```

La super- classe prédéfinie Object

La classe prédéfinit « Object » est la classe parente de toutes les classes Java (super classe dont méthodes sont héritées par toutes les classes). La classe Object dispose de quelques méthodes (utilisées telles quelles, soit être redéfinies) :

- getClass() : renvoie un objet de type Class représentant la classe de l'objet.
- toString() : Elle fournit une chaîne(un objet de type String) contenant le nom de la classe concernée, et l'adresse de l'objet en hexadécimal (précédée de @).
- equals() : implémente une comparaison des références et se contente, ainsi, de comparer les adresses des deux objets concernés.
- finalize() : est appelee quand la machine virtuelle termine son execution.
- clone() : renvoie une copie distincte d'un objet. si l'objet d'origine est ensuite modifie, sa copie ne sera pas affecte.

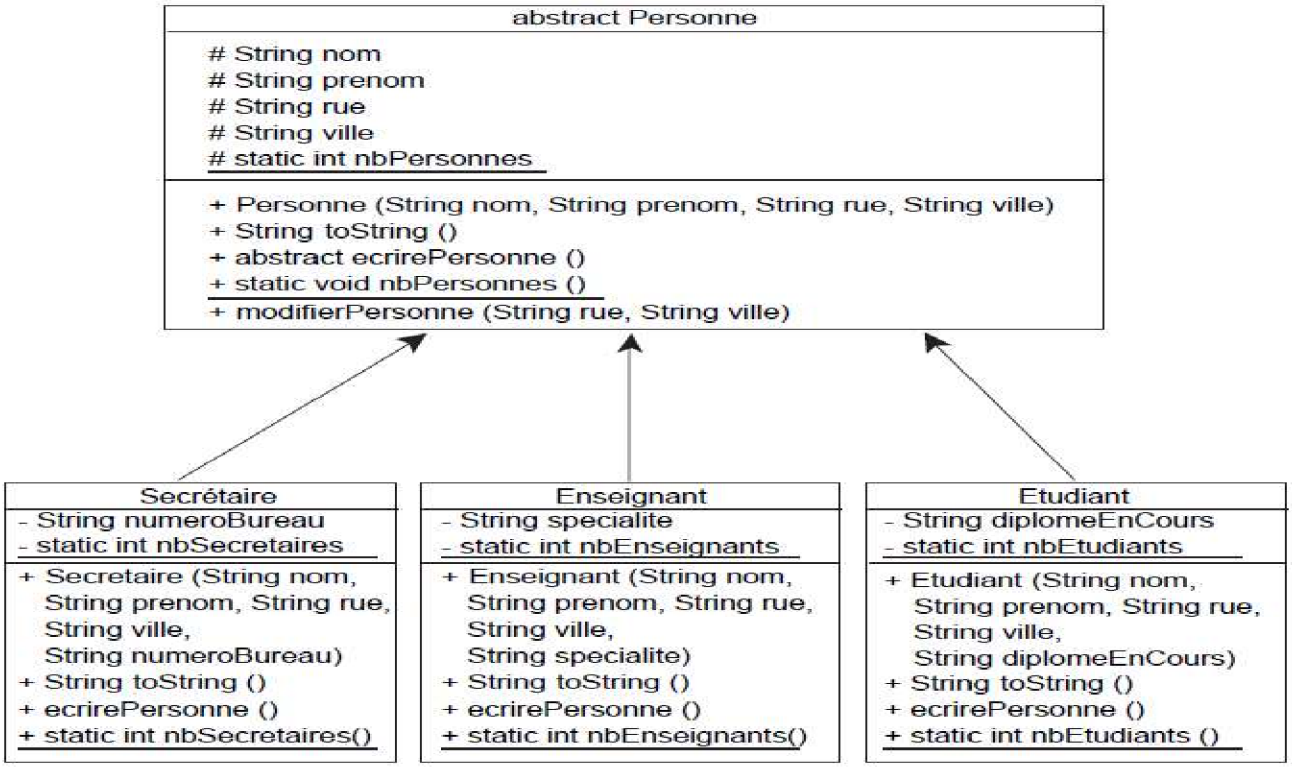
Les classes abstraites

Une classe **abstraite** (utilisation du mot clé **abstract** dans la déclaration) est une classe qui ne permet pas d'instancier des objets. C'est une classe de base qui comporte toutes les fonctionnalités à disposer pour ses descendantes. Les fonctionnalités à définir obligatoirement par ces dernières correspond à des méthodes **abstraites** (utilisation uniquement de la signature) **.**

Exemple :.

```
abstract class A  {
   public void f() { ..... }            // f est définie dans A
   public abstract void g(int n);  //g n'est pas définie dans A (en-tête fournie)
}
class B extends A  {
   …
   public void g(int n){…}  //g est définie dans B
}
```

Exemple de l'hiérarchie d'héritage



Les interfaces

- Une interface définit les en-têtes d'un certain nombre de méthodes (méthodes abstraites à définir) , ainsi que des constants (semblable à une classe abstraite, elle n'est pas instanciée et ne comporte pas de constructeurs)

  Syntaxe de déclaration d'une interface:

  [public] interface nomInterface [extends nomInterface1, nomInterface2...]{

  …

  }

- Une classe implémente une interface lorsqu'elle fournit une définition pour ces méthodes (dans ce cas cette définition ne peut pas être différée)

  Syntaxe d' implémentation:

  Modificateurs class nomClasse [extends superClasse] [implements nomInterface1, nomInterface 2, ...] {

  …

  }

Exemple

```
abstract class Personne {
    private String prenom, nom;
    public Personne(String p,String n){ prenom=p; nom=n; }
    abstract void travailler();  }
public interface Affichage{   public void afficher();  }
public class Enseignant extends Personne implements Affichage{
    private String numSomme;
    public Enseignant(String p,String n ,String d){ super(p,n); numSomme=d; }
    public void travailler(){ System.out.println("travailler pour moi c'est enseigner"); }
    public void afficher(){  System.out.println("Mon DOTI est:"+numSomme); }   }
public class Etudiant extends Personne implements Affichage{
    private String CNE;
    public Etudiant(String p,String n ,String c){    super(p,n); CNE=c; }
    public void travailler(){ System.out.println("travailler pour moi c'est etudier"); }
    public void afficher(){ System.out.println("Mon CNE est:"+CNE); }  }
public class Test{
    public static void main(String[] args){
        Personne[ ] tab = new Personne [2] ;
        tab [0] =new Etudiant("Hamza","Ali","2008/201");
        tab [1] ens=new Enseignant("Hakim", "Omar","65744");
        for (Personne p : tab) { p.afficher(); p.travailler();}  }
}
```


Université Hassan 1er
École Nationale des Sciences Appliquées – Berrechid

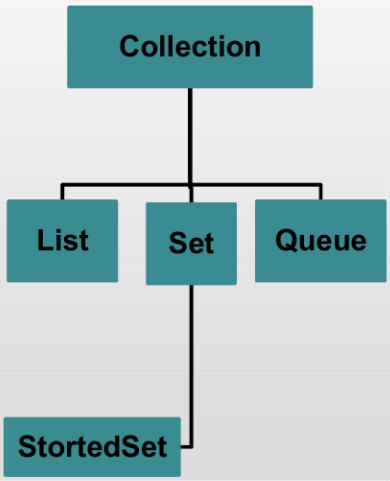
Module : Programmation Orientée Objet (JAVA)

Chapitre 4 : Les collections en JAVA

Pr LAHCEN MOUMOUN
ensablearn@gmail.com

Notion de collection

- Une collection est une structure de données qui rassemble des objets (qualifiés d'éléments de la collection , ils ont un même type) en une seule unité (objet).
- Il existe trois catégories de collections définies dans le paquetage «java.util»:
  - Les listes (List) sont une séquence ordonnée d'objets. On peut trouver plusieurs exemplaires du même objet dans une liste. Un système d'indexation permet l'accès (rapide) aux éléments de la liste.
  - Les ensembles (Set) rassemblent des éléments sans ordre particulier. Un Set n'autorise pas la duplication des données. Les éléments d'un Set ne sont pas accédés par un index. SortedSet est un Set trié
  - les files (Queue) sont les collections accessibles en mode FIFO (First In First Out).

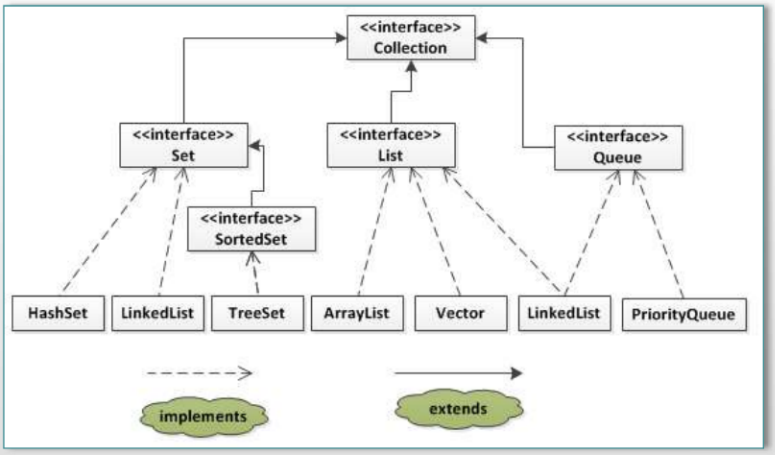


Structure générale d'une classe JAVA

- Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques implémentant l'interface Collection<E>, E représentant le type des éléments de la collection. Tous les éléments de la collection sont , ainsi, de même type E (ou, à la rigueur, d'un type dérivé de E).
- Java offre une architecture unifiée pour représenter et manipuler les collections. Cette architecture est composée de 3 parties :
  - Une hiérarchie d'interfaces permettant de représenter les collections sous forme de types abstraits.
  - Des implémentations de ces interfaces.
  - Implémentation de méthodes (algorithmes) liées aux collections (recherche, tri, etc.).

Implémentation des interfaces

Plusieurs types de collection sont définis comme des interfaces qui sont étendues et implémentées dans le « framework » de collections Java JCF (Java Collections Framework).



- ArrayList (liste de tableaux);
- LinkedList (liste chaînée);
- HashSet (table de découpage);
- TreeSet (arbre).

Création d'un objet collection

- Toute classe collection C<E> dispose d'un constructeur sans argument créant une collection vide :

```
C<E> c = new C<E>() ; // E : type de données des éléments de la collection.
```

- Elle dispose également d'un constructeur recevant en argument une autre collection (n'importe quelle classe implémentant l'interface Collection : liste, vecteur dynamique, ensemble) :

```
/* création d'une collection c2 comportant tous les éléments de c */

C<E> c2 = new C<E> (c) ;
```

Exemple : Pour une liste de String, la création se fera comme suit:

```
List<String> liste = new LinkedList<String>();
```

Les vecteurs dynamiques - classe ArrayList

- La classe ArrayList offre des fonctionnalités d'accès rapide comparables à celles d'un tableau d'objets. Bien qu'elle implémente, comme LinkedList, l'interface List, sa mise en œuvre est différente et prévue pour permettre des accès efficaces à un élément de rang donné(le premier élément est de rang 0, comme dans un tableau usuel).
- En outre, cette classe offre plus de souplesse que les tableaux d'objets dans la mesure où sa taille (son nombre d'éléments) peut varier au fil de l'exécution.

Ajout et suppression d'éléments d'une collection ArrayList

- La classe ArrayList dispose d'une méthode d'ajout supplémentaire **add(i,elem)** qui permet d'ajouter un élément elem en un rang i donné (le nouvel élément prend la place du $i^{ième}$, ce dernier et tous ses suivants étant décalés d'une position).
- La classe ArrayList dispose d'une méthode spécifique **remove** permettant de supprimer un élément de rang donné. Elle fournit en retour l'élément supprimé.

```
ArrayList <E> v ;
.....
/* suppression du ième élément de v qu'on obtient dans o */
E o = v.remove (i) ;
```

- La classe ArrayList possède une méthode spécifique **removeRange** permettant de supprimer plusieurs éléments consécutifs (de n à p) :

```
ArrayList <E> v ;
.....
v.removeRange (n, p) ;        // supprime les éléments de rang n à p de v
```

Accès aux éléments d'une collection ArrayList

- les méthodes d'accès ou de modification d'un élément en fonction de sa position font tout l'intérêt des vecteurs dynamiques.
- On peut connaître la valeur d'un élément de rang i par **get(i)**. Généralement, pour parcourir tous les éléments de type E d'un vecteur v, on procédera ainsi :

```
// depuis le JDK 5.0
for (E e : v) {
    // utilisation de e
}
```

```
// avant JDK 5.0
for (int i=0 ; i<v.size() ; i++)  {
    // utilisation de v.get(i)
}
```

- On peut remplacer par elem la valeur de l'élément de rang i par **set(i, elem).** Cette méthode fournit la valeur de l'élément avant modification.

La classe ArrayList : Exemple

```
import java.util.* ;
public class Array1{
  public static void main (String args[]) {
    ArrayList <Integer> v = new ArrayList <Integer> () ;
    System.out.println ("En A : taille de v = " + v.size() ) ;
    /* on ajoute 10 objets de type Integer */
    for (int i=0 ; i<10 ; i++) v.add (new Integer(i)) ;
    System.out.println ("En B : taille de v = " + v.size() ) ;
    /* affichage par acces direct (get) à chaque element */
     System.out.println ("En B : contenu de v = ") ;
    for (Integer e : v)  System.out.print (e + " ");
    System.out.println () ;
     /* suppression des elements de position donnee */
    v.remove (3) ; v.remove (5) ; v.remove (5) ;
    System.out.println ("En C : contenu de v = " + v) ;
     /* ajout d'elements a une position donnee */
    v.add (2, new Integer (100)) ;  v.add (2, new Integer (200)) ;
    System.out.println ("En D : contenu de v = " + v) ;
     /* modification d'elements de position donnee */
    v.set (2, new Integer (1000)) ; // modification element de rang 2
    v.set (5, new Integer (2000)) ; // modification element de rang 5
    System.out.println ("En D : contenu de v = " + v) ;  }
}
```

Les classes enveloppes *Integer* , *Boolean*, *Character*, *Byte*, *Short*, , *Long*, *Float* et *Double* permettent de manipuler les types primitifs comme des objets. Elle encapsule respectivement des valeurs du type primitif int, *boolean*, *char*, *byte*, *short*, *int*, *long*, *float* et *double*.

Les itérateurs

Les itérateurs sont des objets qui permettent de "parcourir" un par un les différents éléments d'une collection. Cependant, **il ne faut pas modifier la collection pendant qu'on utilise l'itérateur.** Il existe deux sortes d'itérateurs :

- **monodirectionnels** : le parcours de la collection se fait d'un début vers une fin ; on ne passe qu'une seule fois sur chacun des éléments ;
- **bidirectionnels** : le parcours peut se faire dans les deux sens ; on peut avancer et reculer à sa guise dans la collection.

Les itérateurs monodirectionnels : l'interface Iterator

Chaque classe collection dispose d'une méthode nommée **iterator** fournissant un itérateur monodirectionnel, c'est-à-dire un objet d'une classe implémentant l'interface **Iterator<E>**. Associé à une collection donnée, il possède les propriétés suivantes :

- À un instant donné, un itérateur indique une position courante désignant un élément de la collection. On peut obtenir l'objet désigné par un itérateur en appelant la méthode **next** de l'itérateur, ce qui, en outre, avance l'itérateur d'une position.
- La méthode **hasNext** de l'itérateur permet de savoir si l'itérateur est ou non en fin de collection.
- La classe Iterator dispose d'un constructeur recevant un argument entier représentant une position dans la collection (le premier élément d'une collection porte le numéro 0).

Parcours d'une collection avec itérateurs monodirectionnels

- On pourra parcourir tous les éléments d'une collection c<E>, en appliquant le canevas suivant :

```
Iterator<E> iter = c.iterator () ; // renvoie un objet désignant le 1ier élément s'il existe
while ( iter.hasNext() ) {
    E o = iter.next () ;
    // utilisation de o
}
```

Si l'on souhaite parcourir plusieurs fois une même collection, il suffit de réinitialiser l'itérateur en appelant à nouveau la méthode **iterator**.

Méthodes des itérateurs monodirectionnels

- L'interface Iterator prévoit la méthode **remove** qui supprime de la collection le dernier objet (objet courant) renvoyé par next (une exception **IllegalStateException** est déclenché par remove si l'élément courant n'est pas défini).
- La méthode **add** (element), fournit la valeur true lorsque l'ajout a pu être réalisé, ce qui sera le cas avec la plupart des collections, exception faite des ensembles ; dans ce cas, on obtient la valeur false si l'élément qu'on cherche à ajouter est déjà "présent" dans l'ensemble. De la même façon, toute collection dispose d'une méthode **remove** (element) qui supprime un élément de valeur donnée.

Les itérateurs bidirectionnels : l'interface ListIterator

- Certaines collections (listes chaînées, vecteurs dynamiques) peuvent, par nature, être parcourues dans les deux sens. Elles disposent d'une méthode nommée **listIterator** qui fournit un itérateur bidirectionnel implémentant l'interface ListIterator<E> (dérivée de Iterator<E>).
- l'interface ListIterator dispose bien sûr des méthodes **next**, **hasNext** et **remove** héritées de **Iterator**. Mais il dispose aussi d'autres méthodes permettant d'exploiter son caractère bidirectionnel, à savoir : des méthodes **previous**(recule l'itérateur sur la position précédente) et **hasPrevious** (permet de savoir si l'on est ou non en début de collection).

Parcours d'une collection avec itérateurs bidirectionnels

```
ListIterator <E> iter1 = c.ListIterator () ;
while ( iter1.hasNext()) {
    E = iter1.next() ;
    // utilisation de l'objet courant o
}
ListIterator <E> iter ;
iter = l.listIterator (l.size()) ; /* position courante : fin de liste*/
while (iter.hasPrevious()) {
     E o = iter.previous () ;
     // utilisation de l'objet courant o
}
```

Méthodes des itérateurs bidirectionnels

- L'interface **ListIterator** prévoit une méthode **add** qui ajoute un élément à la position courante de l'itérateur. Si ce dernier est en fin de collection, l'ajout se fait tout naturellement en fin de collection. Si l'itérateur désigne le premier élément, l'ajout se fera avant ce premier élément. Par ailleurs, add déplace la position courante après l'élément qu'on vient d'ajouter.
- L'appel **set** (elem) remplace par elem l'élément courant, c'est-à-dire le dernier renvoyé par next ou previous. La position courante de l'itérateur n'est pas modifiée.

Opérations collectives

Toute collection c dispose des méthodes suivantes recevant en argument une autre collection ca :

- **addAll** (ca) : ajoute à la collection c tous les éléments de la collection ca,
- **removeAll** (ca) : supprime de la collection c tout élément apparaissant égal à un des éléments de la collection ca,
- **retainAll** (ca) : supprime de la collection c tout élément qui n'apparaît pas dans la collection ca (on ne conserve donc dans c que les éléments présents dans ca).

Autres méthodes des collections

- La méthode **size** fournit la taille d'une collection, c'est-à-dire son nombre d'éléments
- La méthode **isEmpty** teste si elle est vide ou non.
- La méthode **clear** supprime tous les éléments d'une collection.
- La méthode **contains** (elem) permet de savoir si la collection contient un élément de valeur égale à elem.
- La méthode **toString** est redéfinie dans les collections de manière à fournir une chaîne représentant au mieux le contenu de la collection.
- La méthode **toArray** permet de créer un tableau (usuel) d'objets à partir d'une collection.

Les listes chaînées - classe LinkedList

- La classe **LinkedList** permet de manipuler des listes dites "doublement chaînées". À chaque élément de la collection, on associe les références à l'élément précédent et au suivant. Une telle collection peut ainsi être parcourue à l'aide d'un itérateur bidirectionnel de type ListIterator.
- Le grand avantage d'une telle structure est de permettre des ajouts ou des suppressions à une position donnée (ceci grâce à un simple jeu de modification de références).
- En revanche, l'accès à un élément en fonction de sa valeur ou de sa position dans la liste sera peu efficace puisqu'il nécessitera obligatoirement de parcourir une partie de la liste.

Les listes chaînées - classe LinkedList

La classe LinkedList dispose des méthodes supplémentaires suivantes:

- **getFirst** et **getLast** fournissant respectivement le premier ou le dernier élément de la liste.
- **addFirst** et **addLast** qui ajoutent un élément en début ou en fin de liste.
- **removeFirst** et **removeLast** qui suppriment le premier ou le dernier élément de la liste.

La classe LinkedList : Exemple 1

```
import java.util.* ;
public class Liste1 {
   public static void affiche (LinkedList<String> l){
      ListIterator<String> iter = l.listIterator ();
      while (iter.hasNext()) System.out.print (iter.next() + " ") ; System.out.println () ;  }
   public static void main (String args[]) {
      LinkedList<String> l = new LinkedList<String>() ;
      System.out.print ("Liste en A : ") ; affiche (l) ; l.add ("a") ; l.add ("b") ; // ajouts en fin de liste
      System.out.print ("Liste en B : ") ; affiche (l) ;
      ListIterator<String> it = l.listIterator() ; it.next() ; // on se place sur le premier element
      it.add ("c") ; it.add ("b") ; // et on ajoute deux elements
      System.out.print ("Liste en C : ") ; affiche (l) ;
      it = l.listIterator() ; it.next() ; // on progresse d'un element
      it.add ("b") ; it.add ("d") ;    // et on ajoute deux elements
      System.out.print ("Liste en D : ") ; affiche (l) ; it = l.listIterator (l.size()) ; //on se place à la fin
      while (it.hasPrevious()) {    // on recherche le dernier b
            String ch = it.previous() ; if (ch.equals ("b")) it.remove() ; // et on le supprime
            break ;  }
      System.out.print ("Liste en E : ") ; affiche (l) ;
      it = l.listIterator() ; it.next() ; it.next() ; // on se place sur le deuxieme element
      it.set ("x") ;         // qu'on remplace par "x" System.out.print ("Liste en F : ") ; affiche (l) ;    }
}
```

La classe LinkedList : Exemple 2

```
import java.util.* ;
public class Liste2 {
  public static String lireString (String message){
    System.out.print(message);
    Scanner clav= new Scanner(System.in);  return (clav.next());
  }
  public static void main (String args[]) {
    LinkedList<String> l = new LinkedList<String>() ;
    /* on ajoute à la liste tous les mots lus au clavier */
     System.out.println ("Donnez une suite de mots (vide pour finir)") ;
     while (true) {
        String ch =lireString("Donner chaine (null pour arreter):") ;  if (ch.length() == 0) break ;
        l.add (ch) ;  }
    System.out.println ("Liste des mots à l'endroit :") ;
    ListIterator<String> iter = l.listIterator() ;
    while (iter.hasNext()) System.out.print (iter.next() + " ") ;
    System.out.println ("\nListe des mots a l'envers :") ;
    iter = l.listIterator(l.size()) ; // iterateur en fin de liste
    while (iter.hasPrevious()) System.out.print (iter.previous() + " ") ;
  }
}
```


Les ensembles

- Un ensemble est une collection non ordonnée d'éléments dont aucun élément ne pouvant apparaître plusieurs fois. Pour faciliter le test d'appartenance à un ensemble, deux classes ont été employées, il s'agit de :
  - La classe **HashSet** qui recourt à une technique de hachage, ce qui conduit à une efficacité du test d'appartenance d'une compléxité de l'ordre de **O(1)**,
  - La classe **TreeSet** qui utilise un arbre binaire pour ordonner complètement les éléments, ce qui conduit à une efficacité du test d'appartenance d'une complexité de l'ordre de **O(Log N).**
- Les deux classes HashSet et TreeSet disposent de la méthode iterator prévue dans l'interface Collection. Elle fournit un itérateur monodirectionnel (Iterator) permettant de parcourir les différents éléments de la collection :

```
HashSet<E> e ; // ou TreeSet<E> e
Iterator<E> it = e.iterator() ;
while (it.hasNext()) {
E o = it.next() ;
// utilisation de o
}
```

Ajout et suppression d'éléments d'un ensemble

- La seule façon d'ajouter un élément à un ensemble est d'utiliser la méthode **add** prévue dans l'interface Collection. Elle s'assure que l'élément en question n'existe pas déjà et sera ajouter mais pas obligatoirement en "fin d'ensemble". (il est impossible d'ajouter un élément à une position donnée) :

```
HashSet<E> e ; E elem ;
boolean existe = e.add (elem) ;
```

- Un des grands avantages des ensembles est d'effectuer la suppression d'un élément donnée avec une efficacité en O(1) (pour HashSet) ou en O(Log N) (pour TreeSet). La méthode **remove** de l'itérateur monodirectionnel permet de supprimer l'élément courant .

```
TreeSet<E> e ; E o ;
boolean trouve = e.remove (o) ;
```

- La méthode **contains** permet de tester l'existence d'un élément, avec toujours une efficacité en O(1) ou en O(Log N).

Exemple

```
import java.util.* ;
public class Ens1{
   public static <E> void affiche (HashSet<E> ens)  {
      Iterator<E> iter = ens.iterator () ;
      while (iter.hasNext())  System.out.print (iter.next() + " ") ;
      System.out.println () ;  }
    public static void main (String args[]) {
      int t[] = {2, 5, -6, 2, -8, 9, 5} ;   HashSet<Integer>ens = new HashSet<Integer>() ;
      /* on ajoute des objets de type Integer */
      for (int v : t) {
          boolean ajoute = ens.add(v) ;
          if (ajoute) System.out.println (" On ajoute " + v) ;
          else System.out.println (" " + v + " est deja present") ;  }
      System.out.print ("Ensemble en A = ") ; affiche (ens) ;
      /* on supprime un eventuel objet de valeur Integer(5) */
      Integer cinq = 5 ;
      boolean enleve = ens.remove (cinq) ;
      if (enleve) System.out.println (" On a supprime 5") ;
      System.out.print ("Ensemble en B = ") ; affiche (ens) ;
      /* on teste la presence de Integer(5) */
      boolean existe = ens.contains (cinq) ;
      if (!existe) System.out.println (" On ne trouve pas 5") ;  }
}
```

Utilisation de la méthode compareTo

- Dès que les éléments d'une collection et plus particulièrement ceux d'un ensemble ne sont pas de type String ou enveloppe, il est nécessaire de définir la manière de construction et de comparaison des éléments de la collection.
- Pour les éléments qui sont des objets d'une classe E, Il est possible de définir cette classe en implémentant l'interface Comparable et redéfinir, ainsi, la méthode **compareTo** (« public int compareTo (E o) ») . Celle-ci doit comparer l'objet courant (this) à l'objet o reçu en argument et renvoyer un entier :
  - négatif si l'on considère que l'objet courant est "inférieur" à l'objet o (au sens de l'ordre qu'on veut définir),
  - nul si l'on considère que l'objet courant est égal à l'objet o (il n'est ni inférieur, ni supérieur),
  - positif si l'on considère que l'objet courant est "supérieur" à l'objet o.

Les ensembles TreeSet

- La classe **TreeSet** propose une organisation utilisant un "arbre binaire", lequel permet d'ordonner totalement les éléments. On y utilise la relation d'ordre induite par la méthode compareTo des objets.
- Dans ces conditions, l'utilisation d'un arbre binaire permet de disposer en permanence d'un ensemble totalement ordonné (trié). La classe TreeSet dispose de deux méthodes spécifiques first et last fournissant respectivement le premier et le dernier élément de l'ensemble.
- Dans un ensemble TreeSet, la méthode equals n'intervient ni dans l'organisation de l'ensemble, ni dans le test d'appartenance d'un élément. L'égalité est définie uniquement à l'aide de la méthode compareTo.

Exemple

```
public class Point implements Comparable {
  Point (int x, int y) { this.x = x ; this.y = y ; }
  public int compareTo (Object pp) {
    Point p = (Point) pp ; // egalite si coordonnees egales
    if (this.x < p.x) return -1 ;
       else if (this.x > p.x) return 1 ;
            else if (this.y < p.y) return -1 ;
                 else if (this.y > p.y) return 1 ;
                      else return 0 ;
  }
  public String toString(){
    return("[" + x + " " + y + "] ") ;
  }
  private int x, y ;
}
```


Exemple

```
import java.util.*;
public class TestTreeSet {
    public static void affiche (TreeSet<Point> ens){
        Iterator<Point> iter = ens.iterator() ;
        while (iter.hasNext()) {
            Point p = iter.next() ;
            System.out.print(p.toString()) ;  }
        System.out.println () ;
    }
    public static void main (String args[]){
        Point p1 = new Point (1, 3), p2 = new Point (2, 2) ;
        Point p3 = new Point (4, 5), p4 = new Point (1, 8) ;
        Point p[] = {p1, p2, p1, p3, p4, p3} ;
        TreeSet<Point> ens = new TreeSet<Point> () ;
        for (Point px : p) {
            System.out.print ("le point ") ; System.out.print(px.toString());
            boolean ajoute = ens.add (px) ;
            if (ajoute) System.out.println (" a ete ajoute") ;
                else System.out.println ("est deja present") ;
        }
        System.out.print ("ensemble = " ) ; affiche(ens) ;
     }
}
```

Les ensembles HashSet

- Dans le cas des **HashSet**, on peut définir convenablement:
  - La méthode equals est toujours utilisée pour définir l'appartenance d'un élément à l'ensemble,
  - La méthode hashCode est utilisée pour calculer le code de hachage d'un objet. Elle permet d'ordonnancer les éléments d'un ensemble.
- Si l'on souhaite pouvoir définir une égalité des éléments basée sur leur valeur effective, il va donc falloir définir dans la classe correspondante une méthode hashCode qui doit fournir le code de hachage correspondant à la valeur de l'objet :

```
int hashCode();
```

Dans la définition de cette fonction, il ne faudra pas oublier que le code de hachage doit être compatible avec equals. De même, on ne peut pas se permettre de définir seulement equals sans (re)définir hashCode.

Exemple

```
class Point {
   Point (int x, int y) { this.x = x ; this.y = y ; }
   public int hashCode () {
      return x+y ; }
   public boolean equals (Object pp) {
      Point p = (Point) pp ;
      return ((this.x == p.x) & (this.y == p.y)) ;
   }
   public void affiche () {
      System.out.print ("[" + x + " " + y + "] ") ;
   }
   private int x, y ;
}
```

Exemple

```
import java.util.* ;
public class EnsPt1 {
   public static void affiche (HashSet<Point> ens) {
       Iterator<Point> iter = ens.iterator() ;
       while (iter.hasNext()) {
          Point p = iter.next() ;  p.affiche() ;
       }
       System.out.println () ;   }
    public static void main (String args[]){
       Point p1 = new Point (1, 3), p2 = new Point (2, 2) ;
       Point p3 = new Point (4, 5), p4 = new Point (1, 8) ;
       Point p[] = {p1, p2, p1, p3, p4, p3} ;
       HashSet<Point> ens = new HashSet<Point> () ;
       for (Point px : p) {
           System.out.print ("le point ") ; px.affiche() ;
           boolean ajoute = ens.add (px) ;
           if (ajoute) System.out.println (" a ete ajoute") ;
              else System.out.println ("est deja present") ;
           System.out.print ("ensemble = " ) ; affiche(ens) ;
       }
   }
}
```


Université Hassan 1er
École Nationale des Sciences Appliquées – Berrechid

Module : Programmation Orientée Objet (JAVA)

Chapitre 5: Gestion des exceptions

Pr LAHCEN MOUMOUN
ensablearn@gmail.com

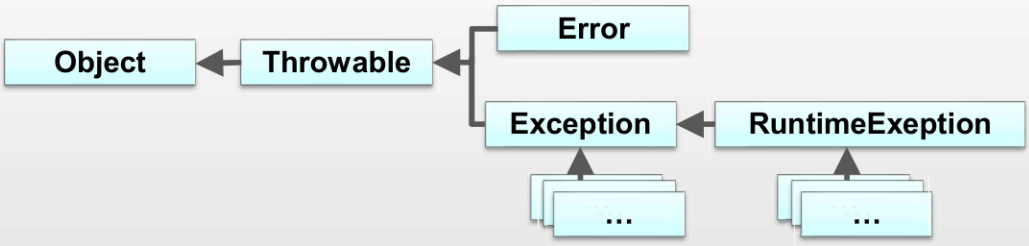
Les exceptions

- Une exception est un événement (une erreur) qui se produit lors de l'exécution d'un programme, et qui va provoquer un fonctionnement anormal (par exemple l'arrêt du programme) de ce dernier. C'est un événement imprévu.

- Java dispose d'un mécanisme très souple nommé gestion d'exception, qui permet de dissocier la détection d'une anomalie de son traitement et de séparer la gestion des anomalies du reste du code, donc de contribuer à la lisibilité des programmes.

Les exceptions

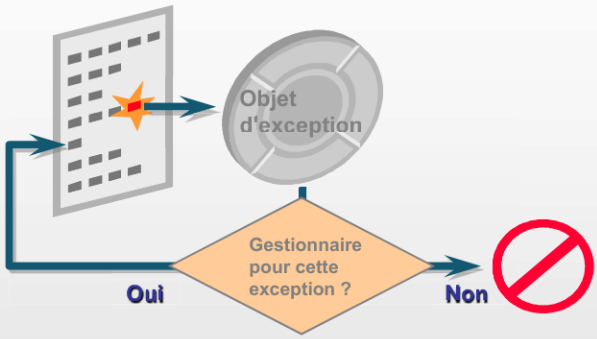
Le langage Java utilise des objets pour représenter les erreurs (exceptions) et l'héritage pour hiérarchiser les différents types d'exception. Chaque exception effective survenant durant l'exécution d'un programme est représentée par une instance de la classe Throwable ou de l'une de ses sous-classes en particulier la classe Exception.



- Error : Situation anormale détectée par la machine virtuelle, jugée non traitable (arrêt instantané de l'application)
- Exception : racine de la hiérarchie des exceptions traitables. Elles correspondent à des erreurs contrôlées (vérifiées et surveillées) par le compilateur avant l'exécution du programme.
- RuntimeException : Situation exceptionnelle détectée par la machine virtuelle mais non contrôlées (unchecked) par le compilateur.

Gestion des exceptions en java

- Lorsque une méthode génère une exception, un gestionnaire récupère et traite l'exception.



- Si la méthode ne contient pas de gestionnaire d'exception capable de traiter l'exception levée; l'interpréteur stoppe l'exécution de la méthode courante et retourne à l'appelant. L'exception se propage, ainsi, vers le haut de la structure lexicale des méthodes Java jusqu'à la méthode main. Si celle-ci ne gère pas l'exception, un message d'erreur s'affiche (avec la trace des appels) et le programme s'arrête.

Capture des exceptions

- Lorsqu'une exception est levée dans une méthode donnée, les instructions qui suivent le lancement de l'exception, se trouvant dans cette méthode, sont ignorées.
- Un bloc try (essayer en anglais) est défini et contient quelques instructions qui risqueraient de lever une ou plusieurs exceptions

```
try { - - - }
```

- Le bloc try est immédiatement suivi du bloc de la définition des différents gestionnaires d'exception. Chaque définition de gestionnaire est précédée d'un en-tête introduit par le mot-clé *catch*

```
catch( TypeException) { - - - }
```

- Si aucun bloc catch ne peut capturer une exception, le traitement de cette exception remonte vers la méthode appelante (on remonte de méthode en méthode), jusqu'à être attrapée (ou bien on est arrivé à la fin du programme).

Exemple

```
import java.util.Scanner;
public class testdiv {
   public static void affiche (String message){
      System.out.print(message); }
   public static int lireInt (String message){
      Scanner  clavier = new Scanner(System.in);
      affiche(message);
      return(clavier.nextInt()); }
   public static int divInt (int n1, int n2) {
      return (n1/n2); }
   public static void main (String [] args) {
      int a,b, c=0;
      a=lireInt("  Entrer le valeur de la 1ier variable: ");
      b=lireInt("  Entrer le valeur de la 2ème variable:");
      c= divInt(a,b);
      System.out.println("  Le resultat de la division est : " + c);
   }
}
```

Scénario n°1:

```
Entrer le valeur de la 1ier variable: 30
Entrer le valeur de la 2ème variable:7
Le resultat de la division est : 4
```

Scénario n°2:

```
Entrer le valeur de la 1ier variable: 20
Entrer le valeur de la 2ème variable:0
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at prjdivzero.testdiv.divInt(testdiv.java:15)
        at prjdivzero.testdiv.main(testdiv.java:20)
```

Scénario n°2:

```
Entrer le valeur de la 1ier variable: a11
```

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at prjdivzero.testdiv.lireInt(testdiv.java:12)
        at prjdivzero.testdiv.main(testdiv.java:18)
```


Exemple

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class testdiv {
   public static void affiche (String message){
      System.out.print(message); }
   public static int lireInt (String message){
      Scanner  clavier = new Scanner(System.in);
      affiche(message); return(clavier.nextInt()); }
   public static int divInt (int n1, int n2) { return (n1/n2); }
   public static void main (String [ ] args) {
      try {
            int a=lireInt("  Entrer le valeur de la 1ier variable: ");
            int b=lireInt("  Entrer le valeur de la 2ème variable:");}
            int c= divInt(a,b);
            System.out.println("  Le resultat de la division est : " + c);
      }
      catch (InputMismatchException e) { affiche("  Erreur de saisie\n"); }
      catch (ArithmeticException e){ affiche("  Division par zéro\n"); }
      System.out.println("  Fin du programme");
   }
}
```

Scénario n°1:

```
Entrer le valeur de la 1ier variable: b22
Erreur de saisie
Fin du programme
```

Scénario n°2:

```
Entrer le valeur de la 1ier variable: 20
Entrer le valeur de la 2ème variable:0
Division par zéro
Fin du programme
```

Gestion des exceptions

- Tous les types d'exception possède les méthodes (hérités de la classe Throwable) suivantes:
  - String getMessage() : Retourne le message de l'exception.
  - String toString(): Retourne une chaine qui contient le type et le message d'exception.
  - void printStackTrace(): Affiche l'erreur (message et type de l'erreur) et la trace de l'exception (propagation et état de la pile d'exécution).
- Lorsqu'une exception est déclenchée dans un bloc try, on recherche parmi les différents gestionnaires associés celui qui correspond à l'exception. L'examen a lieu dans l'ordre où les gestionnaires apparaissent. On sélectionne le premier qui est soit du type exact de l'objet, soit d'un type de base (polymorphisme).

Exemple

```
import java.util.Scanner;
public class testexception {
   public static int lireInt (String message) {
       Scanner  clavier = new Scanner(System.in);
       System.out.print(message);
       return(clavier.nextInt());
   }
   public static void main (String args[]){
       try {
           int t[] ;
           int n = lireInt("taille voulue : ") ;
           t = new int[n] ;
           int i = lireInt("indice : ") ; t[i] = 12 ; System.out.println ("*** fin normale") ;
       }
       catch (NegativeArraySizeException e) {
           System.out.println ("Exception taille tableau negative : " + e.getMessage() ) ;
       }
       catch (ArrayIndexOutOfBoundsException e) {
           System.out.println ("Exception indice tableau : " + e.getMessage() ) ; }
   }
}
```

Scénario n°1:

```
taille voulue : 10
indice : 12
Exception indice tableau : 12
```

Scénario n°2:

```
taille voulue : 10
indice : 5
*** fin normale
```

```
taille voulue : 10
indice : -5
Exception indice tableau : -5
```

Exemple

```
import java.util.Scanner;
public class testdiv {
  public static void affiche (String message){
    System.out.print(message); }
  public static int lireInt (String message){
    Scanner  clavier = new Scanner(System.in);
    affiche(message);return(clavier.nextInt());}
  public static int divInt (int n1, int n2) { return (n1/n2); }
  public static void main (String [ ] args) {
    try {
      int a=lireInt("Entrer le valeur de la 1ier variable: ");
      int b=lireInt("Entrer le valeur de la 2ème variable:");
      int c= divInt(a,b);
      System.out.println("Le resultat de la division est:"+c);
    }
    catch (InputMismatchException e) {
          affiche(e.toString() );} //  on peut utiliser e.printStackTrace( );}
    catch (Exception e) {
          affiche(e.toString() );} //  on peut utiliser e.printStackTrace( );}

    System.out.println("Fin du programme");
  }
```

Scénario n°1:

```
Entrer le valeur de la 1ier variable: 11
Entrer le valeur de la 2ème variable:0
java.lang.ArithmeticException
Fin du programme
```

Scénario n°2:

```
Entrer le valeur de la 1ier variable: C12
java.util.InputMismatchException
Fin du programme
```


Lancement d'une exception

- Pour indiquer qu'une exception s'est produite, on lance ou on lève une exception en instanciant une classe d'exception à l'aide du mot clé throw :

  throw new TypeException() ;

- Pour une méthode susceptible de lever une exception qu'elle ne traite pas localement, cette méthode doit mentionner cette exception dans son en-tête en utilisant, le mot réservé **throws** (une exception peut être traitée localement, ou être propagée par throws).

  modificateur type_retour nom_méthode (parametres) **throws** TypeException1, TypeException2, …{

  - - -

  }

- On peut transmettre une information au gestionnaire d'exception par le biais de l'objet fourni dans l'instruction **throw**.

Exemple

```
import java.util.Scanner;
class Compte {
      public void verser(float mt){  solde=solde+mt;  }
      public void retirer(float mt) throws Exception{
        if(mt>solde) throw new Exception("Solde Insuffisant");
        // on peut utiliser  if(mt>solde) throw new RuntimeException("Solde Insuffisant");
        solde=solde-mt;  }
      public float getSolde(){   return solde;  }
      private int code;  private float solde;
}
public class TestCompte {
      public static float lireFloat (String message) {
        Scanner  clavier = new Scanner(System.in);
        System.out.print(message);  return(clavier.nextFloat());   }
      public static void main(String[] args) {
        Compte cp=new Compte(); float mt1=lireFloat("Montant à verser:");
        cp.verser(mt1);  System.out.println("Solde Actuel:"+cp.getSolde());
        float mt2=lireFloat("Montant à retirer:");
        cp.retirer(mt2); // Le compilateur signale l'Exception
      }
}
```

La classe Exception permet de créer une exception surveillée.
Une exception non surveillée est créer avec la classe RuntimeException

Exemple

```
import java.util.Scanner;
public class TestCompte {
      public static float lireFloat (String message) {
        Scanner  clavier = new Scanner(System.in);
        System.out.print(message);  return(clavier.nextFloat());   }
      public static void main(String[] args) {
        Compte cp=new Compte();
        try {
              float mt1=lireFloat("Montant à verser:");
              cp.verser(mt1);
              System.out.println("Solde Actuel:"+cp.getSolde());
              float mt2=lireFloat("Montant à retirer:");
              cp.retirer(mt2);
        } catch (Exception e) {
              System.out.println(e.getMessage());
        }
        System.out.println("Solde Final="+cp.getSolde());}
}
```

Scénario n°1:

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:2000
Solde Final=3000.0
```

Scénario n°2:

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
Solde Insuffisant
Solde Final=5000.0
```

Personnaliser les exception métier.

- Pour personnaliser une exception on peut définir une classe d'exception qui servira ultérieurement à identifier l'exception concernée et fournir ,ainsi, à l'instruction throw un objet de cette classe. Java impose que la classe d'exception créée dérive de la classe standard Exception.
- On peut transmettre une information au gestionnaire d'exception par l'intermédiaire du constructeur de l'objet exception. Généralement on se transmet un "message" (de type *String*) au gestionnaire. Ce message est retransmis dans le constructeur de la classe d' exception au constructeur de la superclasse *Exception*.
- Toute méthode susceptible de déclencher une exception qu'elle ne traite pas localement doit mentionner son type dans une clause throws figurant dans son en-tête.

Exemple

```
public class MontantNegatifException extends Exception {
     public MontantNegatifException(String message) {  super(message); }
}
public class SoldeInsuffisantException extends Exception {
     public SoldeInsuffisantException(String message) {  super(message);  }
}
public class Compte {
     public void verser(float mt)throws MontantNegatifException {
          if(mt<0) throw new MontantNegatifException("Montant "+mt+" négatif");
          solde=solde+mt;
     }
     public void retirer(float mt)throws SoldeInsuffisantException, MontantNegatifException{
          if(mt<0) throw new MontantNegatifException("Montant "+mt+" négatif");
          if(mt>solde) throw new SoldeInsuffisantException("Solde Insuffisant");
          solde=solde-mt;
     }
     public float getSolde(){  return solde; }
     private int code;
     private float solde;
}
```

Exemple

```
public class TestCompte {
  public static float lireFloat (String message) {
    Scanner  clavier = new Scanner(System.in);
    System.out.print(message);
    return(clavier.nextFloat());
  }
  public static void main(String[] args) throws Exception {
    Compte cp=new Compte();
    try {
      float mt1=lireFloat("Montant à verser:");
      cp.verser(mt1);
      System.out.println("Solde Actuel:"+cp.getSolde());
      float mt2=lireFloat("Montant à retirer:");
      cp.retirer(mt2); }
    catch (SoldeInsuffisantException e) {
      System.out.println(e.toString()); }
    catch (Exception e) {
      System.out.println(e. toString() ); }
    System.out.println("Solde Final="+cp.getSolde());
  }
}
```

Scénario n°1:

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:3000
Solde Final=2000.0
```

Scénario n°2:

```
Montant à verser:y2
java.util.InputMismatchException
Solde Final=0.0
```

Scénario n°3:

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:7000
prjexception.SoldeInsuffisantException: Solde Insuffisant
Solde Final=5000.0
```

Scénario n°4:

```
Montant à verser:5000
Solde Actuel:5000.0
Montant à retirer:-1000
prjexception.MontantNegatifException: Montant -1000.0 négatif
Solde Final=5000.0
```


Le 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.
  - 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.
- D'une manière générale, le bloc finally peut s'avérer précieux dans le cadre de l'acquisition de ressources : 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.

Exercice 1

Que fait ce programme ?

```
class Positif  {
    public Positif (int n) throws ErrConst  {  if (n<=0) throw new ErrConst() ; }   }
class ErrConst extends Exception { }
public class TstPos {
    public static int lireInt (String message){
            Scanner  clavier = new Scanner(System.in);
            affiche(message);return(clavier.nextInt());}
    public static void main (String args[]) {
            System.out.println ("debut main") ;  boolean ok = false ;
            while (!ok) {
               try {
                  int n = lireInt("donnez un entier positif : ") ;
                  Positif ep = new Positif (n) ;
                  ok = true ; }
               catch (ErrConst e) { System.out.println ("*** erreur construction ***") ; }
            }
            System.out.println ("fin main") ;
    }
}
```


Exercice 2

Que fait ce programme ?

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

Module : Programmation Orientée Objet (JAVA)

Chapitre 6: JDBC

Pr LAHCEN MOUMOUN
ensablearn@gmail.com

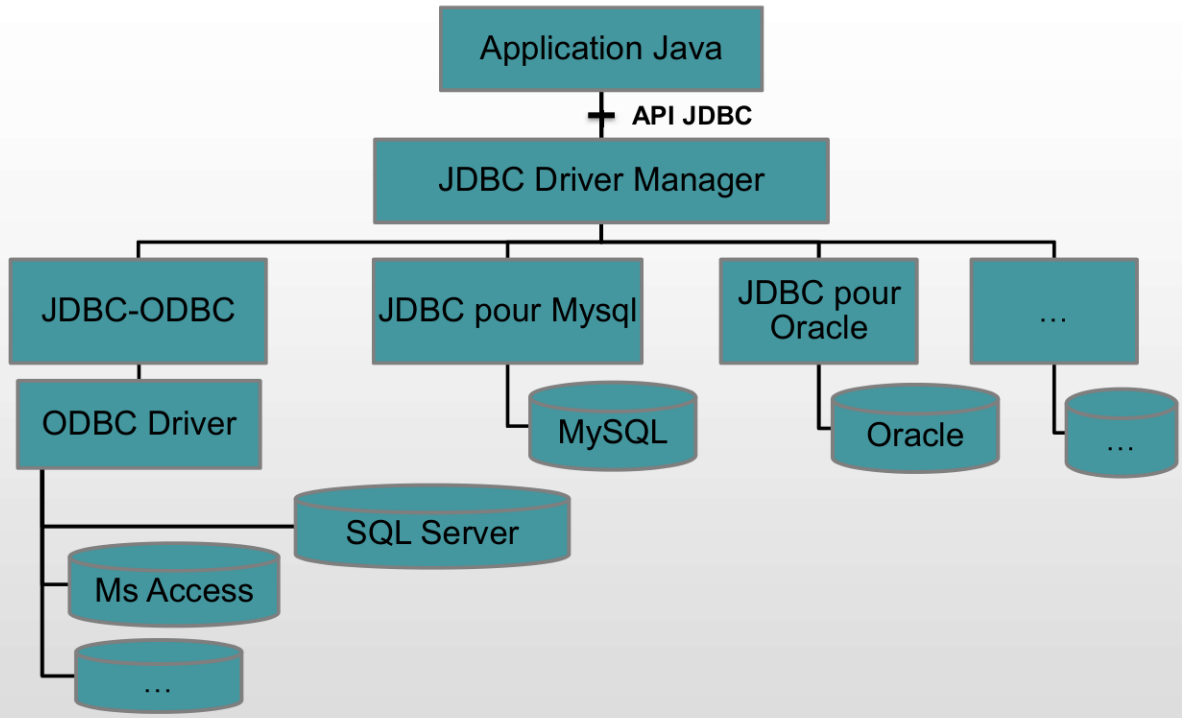
Java et JDBC

- JDBC (Java Data Base Connectivity) est une API Java qui permet d'accéder et manipuler une base de données en exécutant des instructions SQL.
- JDBC offre un accès uniforme une base de données:
  - Portable sur la plupart des OS
  - Indépendant du SGBD (seule la phase de connexion est spécifique – driver)
  - Compatible avec la plus part des SGBDR (Oracle, Postgres, MySQL, Informix, MS SQL Server…)

Pilotes JDBC

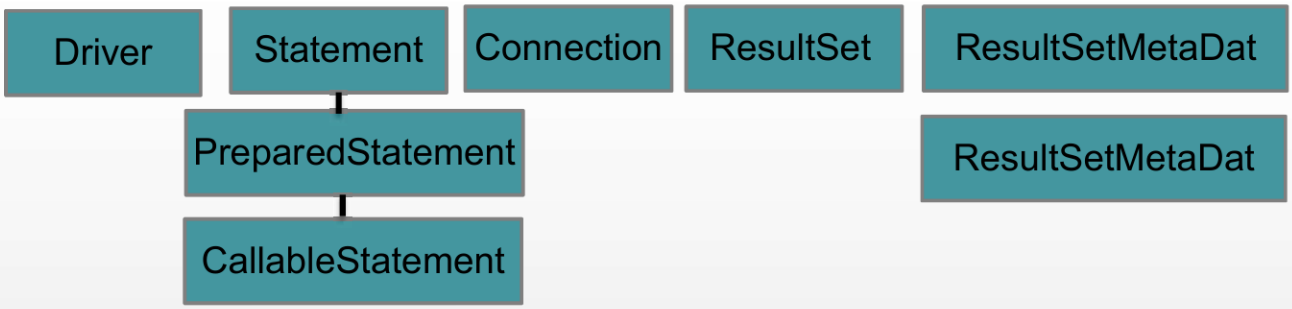
- L'API JDBC fait partie du JDK (Java Devlopment Kit). Elle est représentée par le paquetage java.sql.
- Pour travailler avec un SGBD, il faut disposer de classe qui implémente les interfaces de JDBC. Un ensemble de telles classes est désigné sous le nom de driver JDBC.
- Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java communiquer avec n'importe quelle source de données via les pilotes ODBC (Open Data Base Connectivity)

Pilotes JDBC



- Driver Manager = Gestionnaire de tous les divers chargés par une application Java.
- Un gestionnaire JDBC prend la forme d'un fichier JAR, il faut le référencier et l'ajouter au projet JAVA

Les interfaces du package Java.SQL



- Driver : renvoie une instance de Connection
- Connection : connexion à une base
- Statement : instruction SQL
- PreparedStatement : instruction SQL paramétrée
- CallableStatement : procédure stockée dans la base
- ResultSet : enregistrements récupérés par une instruction SQL
- ResultSetMetaData : description des enregistrements récupérés
- DatabaseMetaData : informations sur la base de données

Exceptions liées aux bases de données

La classe SQLException du paquetage java.sql enrichit la classe Exception pour permettre de mieux gérer les éventuelles erreurs émises par une BD:

- String getMessage(): Envoie un message décrivant l'erreur.
- String getSQLState(): Envoie un code d'erreur associé à une requête SQL.
- int getErrorCode(): Envoie un code d'erreur spécifique au fournisseur du pilote.

Exceptions liées aux bases de données

On procède à la connexion à une base de données en deux étapes :

1. **Chargement du driver par la JVM** : il s'agit d'une implémentation de l'interface **Java.sql.Driver,** le driver peut être chargée en appelant la méthode **forName** de la classe **java.long.Class**

   Exemple

   ```
   Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" ) ;
   Class.forName("com.mysql.jdbc.Driver");
   Class.forName("oracle.jdbc.driver.OracleDriver");
   ```

2. **Connexion :** elle s'effectue en appelant la méthode getConnection de la classe **java.sql.DriverManager.**

   Pour un pilote com.mysql.jdbc.Driver :

   ```
   Connection conn=DriverManager.getConnection("jdbc:mysql:
               //localhost:3306/ DB", "user", "pass" );
   ```

   Le premier paramètre décrit le driver JDBC, la machine où tourne le SGBDR et le nom de la base de données.

Exemple

```
Import java.sql.*; import javax.swing.JOptionPane;
Public class testConecionJDBC{ // BD de test installée par défaut avec MYSQL
  public static void main ( String [] args){
    Connection conn=null;
    try {
      Class.forName("com.mysql.jdbc.Driver"); //Chargement du driver JDBC/MYSQL
      conn=DriverManager.getConnection("jdbc:mysql: //localhost:3306/test");
      JOptionPane.showMessageDialog(null, "Connexion OK" );}
    catch (ClassNotFoundException ex) { // si le driver n'est pas charger par forName
      JOptionPane.showMessageDialog(null, "Classe introuvable"   + ex.getMessage());}
     catch (SQLException ex) {
      JOptionPane.showMessageDialog(null, " Connexion Impossible"  +ex.getMessage());}
    finally {
      try {
         if (conn!=null) conn.close();} //Fermeture de la connexion
      Catch(SQLExeption ex) {ex.printStackTrace();}
    }
    System.exit(0); //arrêt de la JVM;
    }
}
```

Connexion JDBC (java.sql.Connection)

La connexion renvoyée par la méthode getConnction de la classe java.sql.DriverManager permet d'effectuer des opérations sur la base de données. Ces principales méthodes sont:

- createStatement : cette méthode renvoient une instance de java.sql.Statement utilisée pour exécuter une instruction SQL.
- prepareStatement : cette méthode précompilent des instructions SQL paramétrées et renvoient une instance de java.sql.PreparedStatement .
- prepareCall : cette méthode préparent l'appel aux procédures stockées de la BD et renvoient une instance de CallableStatement .
- setAutoCommit, commit, rollback : gèrent les transactions sur la BD.
- getMetaData : cette méthode renvoie une instance de java.sql.DatabaseMetaData pour obtenir des informations sur la base de données et sur ses possibilités.
- Close et isclosed : gèrent la fermeture d'une connexion.

Exécuter des instructions SQL (java.sql.Statement)

Les méthodes **createStatement** d'une connexion renvoient un objet dont la classe implémente l'interface **java.sql.Statement**. Les méthodes de cette interface les plus utilisées sont :

- executeUpdate : elle exécute l'instruction SQL en paramètre sur la BD, pour mettre à jour ses tables ou les enregistrements d'une table.
- executeQuery : elle exécute une sélection SQL se servant de l'instruction Select en paramètre et renvoient une instance de java.sql.ResultSet utilisée pour énumérer les enregistrements recherchés ligne par ligne.
- execute() : elle permet d'exécuter n'importe quelle instruction SQL. Cette méthode retourne un booléen valant true si l'instruction renvoie un objet ResultSet et false sinon (le résultat d'une requête de sélection pourra être obtenu par la méthode getResultSet).

Exploiter les résultats d'une sélection SQL

L'interface java.sql.ResultSet est dotée de deux catégories de méthodes, appelées tour à tour :

- Les méthodes next, first, last, beforeFirst, afterLast, relative, absolute (déplacement sur un enregistrement) sont utilisées pour changer de ligne : ces méthodes renvoient true s'il existe une ligne à la position choisie.
- Les méthodes get… comme getString, getInt, getDate, getObject …, renvoient la valeur d'un des champs d'une ligne. Chacune de ses méthodes existe sous deux formes qui prennent en paramètre l'identificateur ou le numéro d'ordre d'un champ de l'instruction Select (le numéro du premier champ est 1).

Types SQL et Types Java

JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT DOUBLE	double
BINARY VARBINARY LONGVARBINARY	byte[]
CHAR VARCHAR LONGVARCHAR	String

JDBC Type	Java Type
NUMERIC DECIMAL	BigDecimal
DATE	java.sql.Date
TIME TIMESTAMP	java.sql.Timestamp
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

Utilisation du JDBC

- Créer un objet Statement

  Statement statement = conn.createStatement();
- Exécuter une requête (l'objet ResultSet retourné contient les lignes du résultat de la requête)

  String query = "SELECT col1, col2, col3 FROM sometable";

  ResultSet resultSet = statement.executeQuery(query);
- Gérer le résultat

```
while(resultSet.next()) {
    System.out.println(resultSet.getString(1) + " " +
            resultSet.getString(2) + " " + resultSet.getString(3));
}
```

- Fermer la connexion

  connection.close();

Exemple

```
import java.sql.*;
import javax.swing.JOptionPane;

class CalculTotalFactures
{
  public static void main(String[] args)
  {
    Connection connexion  = null;
    Statement  instruction = null;
    ResultSet  resultat  = null;
    try
    {
      Class.forName ("com.mysql.jdbc.Driver");
      connexion = DriverManager.getConnection(
                                    "jdbc:mysql:///test"); 1
      instruction = connexion.createStatement();
      instruction.executeUpdate(
          "CREATE TABLE FACTURE(CLIENT CHAR(50),"
          + " ARTICLE VARCHAR(255), MONTANT DECIMAL(9,2))"); 2
```

Exemple

```
    instruction.executeUpdate(
        "INSERT INTO FACTURE (CLIENT, ARTICLE, MONTANT)"
        + " VALUES ('Thomas Durand', 'CDRx10', '6.35')"); 3
    instruction.executeUpdate(
        "INSERT INTO FACTURE (CLIENT, ARTICLE, MONTANT)"
        + " VALUES ('Sophie Martin', 'PC', '1500')");
    instruction.executeUpdate(
        "INSERT INTO FACTURE (CLIENT, ARTICLE, MONTANT)"
        + " VALUES ('Sophie Martin', 'Imprimante', '120.5')");
    resultat = instruction.executeQuery(
        "SELECT ARTICLE, MONTANT FROM FACTURE"
        + " WHERE CLIENT='Sophie Martin'"); 4
    String articles = "";
    double montantTotal = 0;
    while (resultat.next())
    {
      articles += resultat.getString ("ARTICLE") + " ";
      montantTotal += resultat.getDouble ("MONTANT"); 5
    }
    JOptionPane.showMessageDialog(null,
        "Articles : " + articles
        + "\nMontant total : " + montantTotal + " \u20ac");
    instruction.executeUpdate("DROP TABLE FACTURE");
}
```


Exemple

```
    catch (ClassNotFoundException ex)
    {
      JOptionPane.showMessageDialog(null,
            "Classe introuvable " + ex.getMessage ());
    }
    catch (SQLException ex)
    {
      JOptionPane.showMessageDialog(null,
            "Erreur JDBC : " + ex.getMessage ());
    }
    finally
    {
      try
      {
        if (resultat != null)
          resultat.close();
        if (instruction != null)
          instruction.close();
        if (connexion != null)
          connexion.close();
      }
      catch (SQLException ex)
      {
        ex.printStackTrace ();
      }
    }
    System.exit (0);
  }
}
```

Informations sur la BD (java.sql.DatabaseMetaData)

- La méthode **getMetaData** d'une connexion renvoie un objet dont la classe implémente l'interface **java.sql.DatabaseMetaData**. Les méthodes de cette interface permettent d'obtenir des informations complètes sur la base de données : tables, index, procédures stockées, champs, droits d'utilisation…
- La plupart de informations d'une BD sont renvoyées sous forme d'une instance de **java.sql.ResultSet** utilisée pour énumérer principalement les réponses suivantes:
  - **ResultSet getCatalogs() :** Retourner la liste du catalogue d'informations (avec JDBC-ODBC, on obtient la liste des BDs)
  - **ResultSet getTables**(String catalog, String schema, String tablemask, String types[]): Renvoie la liste des tables (utiliser le mask '%' pour retrouver toutes les tables).
  - **ResultSet getColumns** (catalog, schema, tableNames, columnNames): Retourner une description des colonnes correspondant au TableNames et aux columnNames.

Informations sur la BD (java.sql.DatabaseMetaData)

```
Connection connexion = null;
try {
    Class.forName("com.mysql.jdbc.Driver");
    connexion = DriverManager.getConnection ("jdbc:mysql://localhost/test");
    DatabaseMetaData dmd = connexion.getMetaData();
    ResultSet tables = dmd.getTables(connexion.getCatalog(),null,"%",null);
    System.out.println("TABLE_NAME");
    while(tables.next()){
        String TypeTable = (String) tables.getObject("TABLE_TYPE");
        if (TypeTable.equals("TABLE")){
            System.out.println(tables.getString("TABLE_NAME"));
        }
        }
}
```

Informations sur une requête SQL

- L'objet **ResultsetMetaData** renvoyée par **getMetaData** d'un ResultSet permet de déterminer sa structure (nombre de champs, nom, type et taille de chaque champ)
- Les méthodes permettent de retrouver des informations concernant une table :

Méthodes	Résultat	Type
getColumnCount( )	Nombre de colonne constituant la table	int
getTableName(i)	Nom de la table	String
getColumnName(i)	Renvoie le nom de la colonne	String
getColumnLabel(i)	Fournit le titre suggéré de la colonne	String
getColumnTypeName(i)	Type de la colonne au sens base de données	String
getColumnDisplaySize(i)	Taille de la colonne	int
isNullable(i)	Si la colonne accepte des valeurs nulles	int
isAutoIncrement(i)	Si la colonne s'auto-incrémente	boolean

Informations sur une requête SQL

```
Statement instruction = connexion.createStatement();
ResultSet table = instruction.executeQuery("select * from emp");
ResultSetMetaData infoTable = table.getMetaData();
System.out.println("Nom de la table : " + infoTable.getTableName(1));
System.out.println("Nombre de colonnes : "+infoTable.getColumnCount());
for (int i=1; i<=infoTable.getColumnCount(); i++) {
    System.out.print("------------------------ : ");
    System.out.println("Colonne n°"+i);
    System.out.println("Label : "+infoTable.getColumnLabel(i));
    System.out.println("Nom : "+infoTable.getColumnName(i));
    System.out.println("Type : "+infoTable.getColumnTypeName(i));
    System.out.println("Taille : "+infoTable.getColumnDisplaySize(i));
    System.out.println("Null ? "+infoTable.isNullable(i));
    System.out.println("Auto-incrément ? "+infoTable.isAutoIncrement(i));
}
```

Paramétrer une requête SQL

- Les méthodes **PrepareStatement** d'une connexion précompilent une instruction SQL paramétrée. Ainsi via un objet **PreparedStatement** on optimise les traitements (le SGBD analyse une seule fois la requête).
- Chaque paramètre d'une requête paramétrée est symbolisé par un point d'interrogation (?) remplacé par une valeur lors de l'exécution de l'instruction.
- L'interface **java.sql.PreparedStatement** définit (en plus des méthode de **java.sql.Statement**) les méthodes:
  - Les méthodes void **set[type]** (int ordre, [type] val), comme setString, setInt, …, utilisées pour spécifier la valeur de chaque paramètre de l'instruction SQL précompilée (le premier numéro d'ordre 1).
  - Les deux méthodes sans paramètre **executeUpdate** et **executeQuery**, appelées pour exécuter l'instruction SQL précompilée en remplaçant les ? Par la valeur des paramètres.

Informations sur une requête SQL

```
PreparedStatement stmt=connexion.prepareStatement
                                    ("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 spécifies le premier paramètre de la requête
stmt.setString(2,"Ratan");
int i=stmt.executeUpdate();
System.out.println(i+ " enregistrement inséré");
stmt=connexion.prepareStatement ("select * from emp , dept where
                                    emp.deptno=dept.deptno and dname like ?");
stmt.setString(1,"SALES");
ResultSet rs =stmt.executeQuery();
while(rs.next()){  System.out.println(rs.getString("Ename"));   }
stmt.setString(1,"ESTB");
rs =stmt.executeQuery();
while(rs.next()){  System.out.println(rs.getString("Ename")); }
```

Mapping objet relationnel

- En pratique, on cherche toujours séparer la logique de métier de la logique de présentation. Ainsi, une application est généralement devisée en 3 couches:
  - La couche d'accès au données DAO: partie de l'application qui permet d'accéder aux données qui sont souvent stockées dans une base de données relationnelles.
  - La couche métier : regroupe l'ensemble des traitements que l'application doit effectuée.
  - La couche présentation : s'occupe de la saisie et l'affichage des données.
- Le mapping Objet relationnel consiste faire correspondre un enregistrement d'une table de la BD un objet d'une classe correspondante (classe appelé persistante)