Cours sur Java

I- Généralités

- Tout programme Se fonde sur l'utilisation de **classes**
- Une **classe** est constituée d'un ensemble d'**attributs** et d'un ensemble de **méthodes**
- Un programme construit des **instances** de classe (**objet**)
- On peut ranger les classes selon des ensembles : les **paquetages**. Il y a des règles de visibilité (public, protected, private…) entre les classes et entre les attributs et les méthodes qu'elles contiennent. Ces règles dépendent de l'appartenance ou non à un même paquetage.

Il existe de nombreux paquetages dans le JDK (Java Development Kit)
Vous pourrez en trouver un descriptif dans la documentation fournie en ligne par Sun Microsystems sur le site :

http://java.sun.com/docs

Exemples :

- Le paquetage java.lang contient les classes **les plus centrales du langage**. Il contient la classe Object qui est la **super-classe** ultime de toutes les classes. Une classe qui ne déclare pas d'héritage, hérite par défaut de la classe Object.
- Le paquetage java.util définit un certain nombre de **classes utiles** et est un complément de java.lang.
- Le paquetage java.awt (awt pour **Abstract Window Toolkit**) contient des classes pour fabriquer des **interfaces graphiques**.
- Le paquetage java.applet est utile pour faire des **applets**, applications utilisables à travers le Web.
- Le paquetage java.io contient les classes nécessaires aux **entrées-sorties**.
- Le paquetage java.net fournit une infrastructure pour la **programmation réseau**.

Fichiers sources **.java** (obligatoirement) peuvent contenir plusieurs classes mais au **max 1** seule sera **publique**. S'il y en une : **même nom** que le fichier.

Compilation : .java => .class : **bytecode interprétable** par n'importe qu'elle machine virtuelle java (**JVM**) se conforment à la norme (mais attention à la **version** de Java).
Il existe des JVM pour à peu près tous les types de machines et tous les systèmes d'exploitation => un programme Java est **portable** sans recompilation.

II- Premiers programmes

HelloWorld en Java :

```
class HelloWorld {
     public static void main(String arg[]) {
          System.out.println("Bonjour le monde");
     }
}
```

main est toujours définie ainsi. Elle **doit être définie dans une classe**.
public static : méthode publique de classe (on n'a pas besoin d'objet pour l'appeler).
String : classe gérant les chaînes de caractères du paquetage java.lang
L'argument de la méthode main est un tableau de chaînes de caractères. Les chaînes contenues dans ce tableau sont les arguments envoyés par la ligne de commande, le nom du programme **ne faisant pas partie des arguments (à l'inverse du langage C)**.

System : classe de java.lang qui fournit des **fonctionnalités système indépendantes** de la plate-forme d'exécution.
out : flux de sortie standard.
Conventions de nommage : classes Mots en majuscules, méthodes et attributs mots en majuscules sauf le premier qui est toujours en minuscule.

Deuxième programme :

```
class Somme {
     public static void main(String arg[]) {
             int i, somme = 0;

             for(i=1 ; i<=100 ; i++) somme += i ;
             // affiche le texte suivi de la valeur de la variable somme
             System.out.println("Somme : "+somme) ;
      }
}
```

Similitude avec le C :
- Mêmes structures de contrôles ;
- Mêmes types de bases (à part boolean) ;
- Mêmes opérateurs de comparaison et d'affectation.

III- Définition et utilisation d'une classe

```
class Ecriture
     {
       /*chaine est un attribut de la classe Ecriture,
        qui doit contenir une reference vers une instance
        de la classe String*/
       String chaine = "Encore une fois ";

       //methode de la classe Ecriture
       void ecrire(String autreChaine)
       {
         System.out.print(chaine);
         System.out.println(autreChaine);
```

```
    }
}
```

La classe Ecriture a :

- un attribut (notion proche de variable) : chaine (String)
- une méthode (notion proche de fonction) : ecrire

2 catégories de variables :

de **type primitif** (ex. int) cf exemple précédent.
de **type référence** destinées à recevoir des adresses d'objets.

Attention : une variable de type référence est similaire à un pointeur en C mais on ne peut pas faire d'arithmétique (par exemple incrémenter un pointeur pour parcourir un tableau) avec les références.

Les types primitifs sont :

le type booléen **boolean**, qui n'est pas un type entier et qui peut prendre les valeurs false et true

le type caractère **char** (valeur sur 16 bits pour gérer les caractères asiatiques, arabes…),

les types entiers : **byte** (8 bits), **short** (16 bits), **int** (32 bits) et **long** (64 bits)

les types réels en virgule flottante : **float** (32 bits) et **double** (64 bits)

La valeur par défaut de toute variable ou attribut d'un type "par référence" est la valeur **null**.

Commentaires : idem C++ (/* */ comme en C et // qui commente une ligne) + commentaires javadoc (pour génération automatique de documentation).

```
/* commentaire sur plusieurs lignes comme en C
   deuxième ligne */
int i = 0 ; // commentaire sur une seule ligne – tout ce qui est derrière le // est en commentaire
/** commentaire javadoc
    @param i est un paramètre entier
    @return true ou false */
int methode(int i) {…}
// génère une doc similaire à ce que l'on peut trouver sur le site web de Sun.
```

Utilisation de la classe : **instanciation** de la classe Ecrire :

```
ecrivain = new Ecriture();
```

Création d'un nouvel **objet** de la classe Ecriture (allocation de mémoire pour chacun des attributs de la classe et initialisation – ici chaine). new Ecriture(); renvoie une référence qui est stockée dans la variable ecrivain.

Les **parenthèses** qui apparaissent dans l'instanciation s'expliquent par le fait que l'on fait appel à la méthode **Ecriture()** de la classe **Ecriture** appelée le "**constructeur**" de la classe. Toute classe possède un constructeur. Lorsque le programmeur n'a pas mis explicitement un constructeur, comme c'est le cas ici, le compilateur en **ajoute automatiquement un**. Dans le cas de la classe Ecriture (pas de constructeur hérité ; cette notion sera développée ultérieurement), le constructeur ajouté est une méthode qui ne fait rien.

Lorsqu'on instancie une classe, on le fait toujours comme ici à l'aide d'un constructeur de la classe.

```
class PourFeliciter
     {
       public static void main(String[] arg)
       {
         Ecriture ecrivain;

         ecrivain = new Ecriture();
         ecrivain.ecrire("bravo");
         ecrivain.chaine = "et pour finir ";
         ecrivain.ecrire("au revoir");
       }
     }
```

Note : Appel de méthode : variable.methode(parametres);
Accès aux attributs : variable.attribut

IV- Surcharge de méthodes

Java autorise la surcharge de méthodes. C'est à dire qu'on peut avoir plusieurs méthodes de même nom à condition que l'ordre et le type des paramètres soient différents.

Exemple :

```
class Ecriture
     {
       String chaine = "Encore une fois ";
       void ecrire(String autreChaine)
       {
         System.out.print(chaine);
         System.out.println(autreChaine);
       }
       void ecrire(int unEntier)
       {
         System.out.print(chaine);
         System.out.println(unEntier);
       }
     }
```

A l'exécution java choisira la bonne méthode en fonction des paramètres.

V- Les constructeurs

```
class Incremente
{
  int increment;
  int petitPlus;

  Incremente(int increment, int petit)
  {
    this.increment = increment;
    petitPlus = petit;
  }
  Incremente(int increment)
  {
    this.increment = increment;
    petitPlus = 1;
  }

  int additionne(int n)
  {
     return (n + increment + petitPlus);
  }
}

class UtiliseIncremente
{
  public static void main(String[] arg)
  {
    Incremente monAjout = new Incremente(10, 2);
    System.out.println(monAjout.additionne(5));
    Incremente monAjout2 = new Incremente(10);
    System.out.println(monAjout.additionne(5));
  }
}
```

L'exemple utilise une classe possédant des **constructeurs explicites** (le choix du constructeur sera fait en fonction de l'ordre des paramètres comme pour la surcharge).
Un constructeur ne **retourne pas de valeur** mais ne mentionne pas **void** au début de sa déclaration.
Par ailleurs, un constructeur doit posséder le même nom que celui de sa classe.
Il peut posséder des arguments qui seront initialisés de façon classique.
Les constructeurs de la classe Incremente possèdent un paramètre qui a pour nom increment : celui-ci **cache l'attribut** increment de la classe Incremente. Le mot réservé **this** représente, au moment de l'exécution, l'instance de la classe sur laquelle le code est entrain de s'appliquer ; this.increment représente donc l'attribut increment de la classe. L'instruction :

this.increment = increment ;

initialise l'attribut increment de la classe avec la valeur donnée en argument du constructeur. Le problème ne se pose pas avec l'attribut petitPlus et l'argument petit qui ont des noms différents.

VI- Attributs et méthodes de classe (static)

```
class TestStatic
{
```

```
  private String nom;
  static private int nbInstances;

  public TestStatic(String nom)
  {
    this.nom = nom; nbInstances++;
  }
  static {
     nbInstances = 0 ;
  }
  public static int getNbInstances() { return nbInstances; }

  public static void main(String[] argv)
  {
    TestStatic t = new TestStatic("Un");
    TestStatic t1 = new TestStatic("Deux");

    System.out.println(TestStatic.getNbInstances());
    System.out.println(TestStatic.nbInstances);
  }
}
```

Une classe peut posséder des attributs et des méthodes qui existent indépendamment de toute instance. Ils sont déclarés avec le mot clé **static**. On les appelle attributs et méthodes de classe par opposition aux autres attributs et méthodes appelés attributs et méthodes d'instance car ils s'appliquent à une instance particulière.
On y accède en les prefixant par le nom de la classe suivi d'un point.
Ils servent en général à stocker et travailler sur des informations de l'ensemble des instances (ici le nombre d'instances, on peut aussi créer un vecteur de toutes les instances…) ou des méthodes et attributs dont on a besoin avant création d'objets (ici là méthode**main**).

L'exemple utilise aussi les mot-clés **public** / **private** : public ou privé à la classe. Les attributs privés ne sont pas accessible à l'extérieur de la classe (par exemple par une autre classe). Nous y reviendrons quand nous aborderons les packages.

VII- Les tableaux

Les tableaux ne sont ni des objets, ni des types primitifs. Un tableau se rapproche néanmoins d'un objet par le fait :

- qu'il est manipulé par référence
- qu'il nécessite en général un "new" pour être défini.

```
class TableauA {
  public static void main(String[] argv) {
    char[] tableau;

    tableau = new char[2];
    tableau[0] = 'h';
    tableau[1] = 'a';
    System.out.println(tableau);
  }
}
```

On obtient à l'exécution :
ha

On peut voir ici une première façon de déclarer et d'allouer un tableau.
    char[] tableau;
déclare le tableau tableau, et aurait d'ailleurs aussi pu s'écrire comme en C :
    char tableau[];

Par ailleurs, tableau étant déjà défini :
    tableau = new char[2];
alloue un tableau pour deux variables de type char.

On pourra remarquer l'utilisation de la méthode println pour un tableau de caractères.

```
class TableauB {
  public static void main(String[] argv) {

    boolean tableau[]={true,false,true};

    System.out.println("Deuxieme element de tableau : "
                       + tableau[1]);
  }
}
```

Voici une façon de **déclarer et d'allouer** un tableau pour trois variables de type **boolean**.

Vous pouvez remarquer que la concaténation entre une chaîne de caractères et une variable booléenne, effectuée grâce à l'opérateur "+" est possible. En fait, cela revient à concaténer deux chaînes de caractères car la valeur de la variable booléenne est convertie en une chaîne de caractères par l'opérateur de concaténation.

On a la possibilité de connaître la **longueur** d'un tableau référencé par la variable tableau en utilisant : **tableau.length**;
length est en quelque sorte un attribut du tableau, attribut que l'on peut d'ailleurs uniquement lire.

L'exemple suivant a pour objectif de montrer comment manipuler un tableau d'objets, ici un tableau d'Integer. A part pour cela, il n'est pas utile, pour additionner quelques entiers, d'utiliser la classe Integer.

```
class TableauD {
  public static void main(String[] argv) {
    Integer tableau[]= new Integer[4];
    int somme=0;

    for (int i =0;i<tableau.length;i++)
             tableau[i]=new Integer(i);

    for (int i =0;i<tableau.length;i++)
             somme+=tableau[i].intValue();

    System.out.println("Somme des entiers : " +somme);
```

```
   }
}
```

On obtient à l'exécution :
Somme des entiers : 6

Integer : la classe java.lang.Integer, à consulter comme beaucoup d'autre classes du JDK directement dans la documentation de java, contient beaucoup de méthodes statiques ou non, pour gérer le type int ; à titre d'exemple, la méthode public static int parseInt(String s) convertit la chaîne s en un int.
Le constructeur public Integer(int value) crée une instance de la classe Integer ayant pour attribut (privé) un int de valeur value.

tableau[i]=new Integer(i) : il faut noter que **instancier un tableau de trois Integer ne dispense pas de construire une instance d'Integer pour chaque composante du tableau**.

tableau[i].intValue(): la méthode utilisée ici est une **méthode d'instance** de Integer ; elle retourne l'attribut de type int contenu dans l'instance d'Integer invoquée, ici **tableau[i]**.

VIII- Héritage

L'utilisation de l'héritage fait partie intégrante des langages à objets.
Les quelques exemples suivants devraient vous permettre d'acquérir les notions principales liées à l'héritage en Java.

Soit la classe Rectangle définie ainsi :

```
class Rectangle {
        double largeur, hauteur;
        Rectangle(double initL, double initH){
                largeur=initL;
                hauteur=initH;
        }
        double perimetre() {return 2*(largeur+hauteur);}
        double surface() {return largeur*hauteur;}
        double diagonale() {
                return Math.sqrt(largeur*largeur+hauteur*hauteur);
        }
        void doubler() {largeur*=2; hauteur*=2;}
}
```

Si on veut définir une classe Carré, au lieu de tout redéfinir on peut simplement écrire :

```
class Carre extends Rectangle {
     Carre(double cote) {
          largeur = hauteur = cote;
     }
```

```
}
```
Les méthodes de la classe Rectangle seront disponibles automatiquement dans la classe Carré.
Donc on peut écrire :

```
Rectangle r ;
Carre c ;
r = new Rectangle(3,10);
c = new Carre(4);
System.out.println("perimetre du rectangle :"+r.perimetre());
System.out.println("perimetre du carre :"+c.perimetre());
…
```

La classe Carre étend la classe Rectangle. On dit aussi, et cela a la même signification, que la classe Carre hérite de la classe Rectangle. On dit encore que la classe Rectangle est la super-classe de la classe Carre. On dit enfin que la classe Carre est une sous-classe de la classe Rectangle.
On peut si on le veut ajouter des attributs et des méthodes supplémentaires à la classe Carre.

Un héritage peut aussi s'exprimer en disant qu'un Carré est un Rectangle (le verbe être étant utilisé pour représenter l'héritage). Et ainsi on peut écrire la chose suivante :

```
Rectangle r ;
r = new Carre(4);
System.out.println("perimetre du rectangle :"+r.perimetre());
…
```

On peut utiliser le mot clé **super** pour faire référence à la super-classe ainsi :

```
class Carre extends Rectangle {
     Carre(double cote) {
          super(cote,cote); // appel du constructeur de Rectangle
     }
     double perimetre() {
          // on appelle la méthode de même nom de Rectangle
          return super.perimetre();
     }
     public String toString() {
          return "Carré de côté : "+largeur;
     }
}
```

NB : on a aussi surchargé ici la méthode **toString** de la classe **Object**.
En effet, en java, toute classe dérive de la classe Object. Si on ne précise pas le mot-clé extends, implicitement on dérive de cette classe (ici Rectangle dérive donc implicitement d'Object).
La classe Object contient une méthode : public String toString(); Cette méthode est utilisée en particulier par :

- **l'opérateur de concaténation "+"** entre chaînes de caractères ; lorsque un opérande de cet opérateur n'est pas une chaîne de caractères mais un objet, la méthode toString est appliquée à cet objet. Si la méthode toString n'est pas définie dans la classe A de l'objet en question, l'interpréteur cherche dans la classe B dont hérite A. Si la méthode toString n'est pas définie dans la classe B, l'interpréteur cherche dans la classe C dont hérite B et ainsi de suite. Rappelons qu'au sommet de la hiérarchie des classes se trouve la classe Object. Le procédé aboutit donc toujours.
- la méthode : public **void println(Object x)** de la classe **java.io.PrintStream**, qui utilise la méthode toString de la classe de x pour écrire celui-ci, en effectuant en fait **println(x.toString());** (avec la même technique de remontée que ci-dessus pour la concaténation).

Il peut donc être pratique de redéfinir la méthode **toString** lorsqu'on crée une nouvelle classe. On doit attribuer le modificateur public pour redéfinir cette méthode car on ne peut pas redéfinir une méthode en diminuant son niveau de visibilité. Nous reviendrons sur les niveaux de visibilité.

En Java une classe ne peut hériter que d'une seule classe. On parle d'héritage simple.

IX- Méthodes et classes Abstraites

Parfois il peut arriver que l'on ne sache pas écrire une méthode mais qu'on veuille la déclarer pour les classes descendantes. On dira dans ce cas que la méthode est abstraite et on la notera avec le mot-clé **abstract**. Une classe qui a au moins une méthode abstraite sera déclarée abstraite et il sera impossible d'instancier des objets de cette classe.

Par exemple, si on veut définir des classes Disque, Triangle… on pourra avoir pour chacune de ces classes des méthodes périmètre et surface. Donc on peut imaginer vouloir définir une classe GeometriePlane de la façon suivante :

```
abstract class GeometriePlane{
     public abstract double perimetre();
     public abstract double surface();
}
```

Ensuite, les autres classes dériveront de GeometriePlane en définissant les méthodes.

```
class Rectangle extends GeometriePlane {
     ...
     public double perimetre() {
          return 2*(largeur+hauteur);
     }
```

```
      public double surface(){
            return largeur*hauteur;
      }
...
}
class Disque extends GeometriePlane {
     static final double PI = 3.1459;
      double diametre;
      ...
      public double perimetre() {
            return PI*diametre;
      }
      public double surface() {
            return (PI*diametre*diametre)/4;
      }
}
```

On peut voir ci-dessus une utilisation du mot-clé **final** qui permet de définir des constantes. Les constantes sont en général définies static de façon à ce qu'il n'y ait pas une constante par objet.

NB : on peut très bien avoir dans une classe abstraite des attributs et des méthodes non abstraites.

Si on écrit GeometriePlane g = new GeometriePlane(); on obtient une erreur à la compilation puisque la classe est non instaciable.
Par contre on peut écrire : GeometriePlane g = new Disque(); et manipuler le disque comme une instance de la classe GeometriePlane.

X- Interfaces

Une classe dont toutes les méthodes sont abstraites peut aussi être définie sous la forme d'une interface. L'intérêt est qu'une classe peut implémenter (c'est le terme qui est utilisé à la place d'hériter dans ce cas là) plusieurs interfaces.
Les interfaces peuvent dériver les unes des autres. Les interfaces peuvent aussi définir des constantes.

```
interface Geometrie{
      public static final double PI=3.14159;
}
interface Courbe extends Geometrie{
      public double longueur();
      public void doubler();
}
interface Surface extends Geometrie{
      public double surface();
}
```

Les classes Rectangle et Disque peuvent ensuite être décrites de la façon suivante :

```
class Rectangle extends Object implements Courbe, Surface {
      ...
      public double perimetre() {
```

```
            return 2*(largeur+hauteur);
      }
      public double surface(){
            return largeur*hauteur;
      }
...
}
class Disque extends Object implements Courbe, Surface {
     static final double PI = 3.1459;
      double diametre;
      ...
      public double perimetre() {
            return PI*diametre;
      }
      public double surface() {
            return (PI*diametre*diametre)/4;
      }
}
```

Les interfaces sont considérées comme des types (mais ne sont pas instanciables) et on pourra donc écrire la chose suivante :

```
Courbe c;
c = new Disque();
c.perimetre();
```

Par contre : `c = new Courbe();` provoque une erreur de compilation.

En résumé, une classe peut dériver/hériter d'une seule classe et implémenter un nombre quelconque d'interfaces.

XI- Polymorphisme

Le fait de faire hériter plusieurs classes d'une même classe (ou implémenter la même interface) nous permet d'utiliser le polymorphisme pour traiter uniformément un ensemble d'objet de différentes classes.

```
class TestRectDisk7{
  public static void main (String args[]) {
      Courbe c[]=new Courbe[2];
      Surface s[]=new Surface[2];
      Rectangle r =new Rectangle();
      Disque d = new Disque();
      // un rectangle peut être vu comme une courbe ou une surface
      c[0]=r; s[0]=r;
      // idem pour un disque
      c[1]=d; s[1]=d;
      //surface de tous les objets
      double surfTotal=0;
      for (int i=0;i<2;i++)surfTotal+=s[i].surface();
      //Perimetre de tous les objets
```

```
        double periTotal=0;
        for (int i=0;i<2;i++)periTotal+=c[i].perimetre();
  }
}
```

C'est toujours la méthode définie dans la classe qui est réellement instanciée (Rectangle ou Disque) qui est appelée (ou dans la classe ancêtre la plus proche si la méthode n'est pas définie dans la classe d'instanciation).

XII- Paquetages et visibilité

Le concept de paquetage (package en anglais) permet de réunir plusieurs classes dans une même unité logique qui sera représentée par un répertoire sur le disque dur.
On peut aussi définir des sous-paquetages qui correspondent à des sous répertoires.
Dans chaque fichier il faudra écrire : package nom_de_paquetage ; (cette instruction doit être la première du fichier).
Exemple :

```
package geo;
public interface Geometrie{
      public static final double pi=3.14159;
}
```

Ou dans le cas d'un sous-paquetage geo d'un paquetage projet :

```
package projet.geo;
```

Pour utiliser une classe d'un paquetage à l'intérieur d'un autre paquetage on devra soit :

- préfixer le nom de la classe par le nom du paquetage :

```
geo.Geometrie g;
```

- utiliser une directive import :

```
import geo.Geometrie;
...
Geometrie g;
```

Les directives import sont placées après la directive package (s'il y en a une) et avant les définitions de classes. On peut utiliser le caractère * pour dire qu'on veut importer toutes les classes et interfaces d'un package :

```
import geo.*;
...
Geometrie g;
```

Il existe des mots-clés qui permettent de limiter la visibilité des classes, interfaces, méthodes et attributs. Ils sont au nombre de trois (private, protected et public).

Le principe est le suivant :

- Un attribut ou une méthode déclarée private n'est visible que dans sa classe.
- Un attribut ou une méthode déclarée protected n'est visible que dans sa classe, dans les classes du même package et dans les classes dérivées (même si elles ne sont pas dans le même package).
- Un attribut ou une méthode déclarée sans mot-clé (ni private, ni protected, ni public) n'est visible que dans le paquetage dans lequel est définie sa classe.
- Un attribut ou une méthode déclarée publique est visible de toute classe.

D'autre part :

- Une classe déclarée publique est visible dans tout paquetage.
- Une classe qui n'est pas déclarée publique n'est visible que dans son paquetage.

XIII- Exceptions

Attraper une exception

Une exception correspond à un événement anormal ou inattendu. Les exceptions sont des instances de sous-classes des classes `java.lang.Error` (pour des erreurs graves, qui devront généralement conduire à l'arrêt du programme) et`java.lang.Exception` (pour des événements inattendus, qui seront souvent traités de sorte qu'elle ne provoque pas l'arrêt du programme).
Un grand nombre d'exceptions sont définies dans l'API et sont "lancées" par des méthodes de l'API.
Les exemples suivants montreront comment on définit ou comment on lance une exception. Nous nous contentons ici d'en attraper une (une instance de `java.lang.NumberFormatException`), et d'en laisser passer une autre (une instance de`java.lang.ArithmeticException`).

Lorsqu'une exception est lancée, elle va se propager de la façon décrite ci-dessous. Nous supposons que l'exception est lancée par une certaine instruction `I` d'une méthode de nom `uneMethode`. L'explication que nous donnons est récursive.

- Soit l'instruction `I` se trouve dans un bloc de `uneMethode` :
  - précédé du mot réservé **`try`**, bloc que nous appelons "bloc **`try`**"
  - et suivi d'un bloc précédé du mot réservé **`catch`**, bloc que nous appelons "bloc **`catch`**", le mot **`catch`** étant assorti d'un argument, cet argument étant de la classe ou d'une super-classe de l'exception lancée.

  Alors :
  - les instructions qui suivent le lancement de l'exception et intérieures au bloc **`try`** sont ignorées
  - les instructions du bloc **`catch`** sont effectuées
  - le programme reprend normalement avec l'instruction qui suit le bloc **`catch`**.
- Soit l'instruction `I` n'est pas située comme indiqué ci-dessus. Alors, la méthode `uneMethode` se termine . Si `uneMethode` est la méthode `main`, le programme se termine et l'exception n'a pas été attrapée. Sinon, on se retrouve dans la méthode qui a appelée `uneMethode`, au niveau de l'instruction `I'`qui a fait appel à `uneMethode`. L'instruction `I'` lance à son tour l'exception.

Si une exception est lancée et pas attrapée, et donc qu'elle provoque la terminaison du programme, la pile des méthodes traversées par l'exception est indiquée à l'utilisateur.

Une méthode susceptible de lancer une exception sans l'attraper (c'est-à-dire contenant une instruction susceptible de lancer une exception sans que celle-ci soit attrapée à l'intérieur de la méthode) doit l'indiquer dans son en-tête, nous verrons la syntaxe correspondante dans un prochain exemple. Néanmoins, on est dispensé d'indiquer la possibilité de lancement des erreurs les plus courantes, comme par exemple `java.lang.ArrayIndexOutOfBoundsException`.

```
class ExceptionCatch
{
  static int moyenne(String[] liste)
  {
    int somme = 0,entier, nbNotes = 0;

    for (int i = 0; i < liste.length;i++)
      {
        try
          {
            entier = Integer.parseInt(liste[i]);
            somme += entier;
            nbNotes++;
          }
        catch (NumberFormatException e)
          {
            System.out.println("La "+(i+1)+" eme note "+
                               "n'est pas entiere");
          }
      }
    return somme/nbNotes;
  }

  public static void main(String[] argv)
  {
    System.out. println("La moyenne est " + moyenne(argv));
  }
}
```

Pour :

`Java ExceptionCatch ha 15 12` on obtient en sortie

```
La 1 eme note n'est pas entiere
La moyenne est 13
```

et pour : `java ExceptionCatch ha 15.5`

```
La 1 eme note n'est pas entiere
La 2 eme note n'est pas entiere
java.lang.ArithmeticException: / by zero
        at ExceptionCatch.moyenne(ExceptionCatch.java:22)
        at ExceptionCatch.main(ExceptionCatch.java:27)
```

Pour accéder au programme.

Définir sa propre exception

Si on veut pouvoir signaler un événement exceptionnel d'un type non prévu par l'API, il faut **étendre** la classe java.lang.Exception ; la classe étendue ne contient en général pas d'autre champ qu'un (ou plusieurs) **constructeur**(s) et éventuellement une redéfinition de la méthode **toString**. Lors du lancement de l'exception, (à l'aide du mot réservé **throw** comme nous le verrons dans le prochain exemple), on crée une instance de la classe définie.
Voilà une classe héritant de la classe Exception que nous utiliserons dans les deux exemples suivants.

```
class ExceptionRien extends Exception {
  public String toString() {
       return("Aucune note n'est valide");
  }
}
```

Pour cet exemple et le suivant, l'objectif est de calculer une moyenne de notes entières envoyées en arguments par la ligne de commande. Les arguments non entiers (et donc erronés) doivent être éliminés.

On utilise l'exception ExceptionRien.

```
class ExceptionThrow {
  static int moyenne(String[] liste) throws ExceptionRien {
    int somme=0,entier, nbNotes=0;
    int i;

    for (i=0;i < liste.length;i++) {
        try {
            entier=Integer.parseInt(liste[i]);
            somme+=entier;
            nbNotes++;
        }
        catch (NumberFormatException e) {
            System.out.println("La "+(i+1)+" eme note n'est "+
                               "pas entiere");
        }
    }
    if (nbNotes==0) throw new ExceptionRien();
    return somme/nbNotes;
  }

  public static void main(String[] argv) {
    try {
        System.out.println("La moyenne est "+moyenne(argv));
    }
    catch (ExceptionRien e) {
         System.out.println(e);
    }
  }
}
```

Pour : java ExceptionThrows ha 15.5, on obtient :

La 1 eme note n'est pas entiere
La 2 eme note n'est pas entiere
Aucune note n'est valide

throws ExceptionRien : la méthode moyenne indique ainsi qu'il est possible qu'une de ses instructions envoie une exception de type ExceptionRien sans que celle-ci soit attrapée par un mécanisme try-catch. Il est obligatoire d'indiquer ainsi un éventuel lancement d'exception non attrapée, sauf pour les exceptions les plus courantes de l'API. Si vous oubliez de signaler par la clause **throws** l'éventualité d'un tel lancement d'exception, le compilateur vous le rappelera.

throw new ExceptionRien() : on demande ainsi le lancement d'une instance de ExceptionRien. Une fois lancée, l'exception se propagera comme expliqué dans l'exemple précédent. Ici, il y aura sortie de la méthode moyenne, on se retrouve alors à l'appel de la méthode moyenne dans le main, appel qui se trouve dans un "bloc try" suivi d'un "bloc catch" qui attrape les ExceptionRien(s) : l'instruction de ce "bloc catch" est effectuée. Le programme reprend alors son cours normal (pour se terminer).