Outils de développement,

programmation événementielle et IHM

Chapitre 3 : Java, langage de développement objet

Cyrille Bertelle - UFRST Le Havre

0-0

3.1 Présentation de Java

- Applications indépendantes des machines et de leur système d'exploitation ;
- Applications structurellement distribuées sur des réseaux ;
- Langages objets purs : rien ne peut exister en dehors des classes.

Historique

- Début 1990, langage OAK pour domotique (télévision interactive) ;
- codes peu volumineux, efficaces et indépendants de l'architecture ;
- Développement d'Internet et du Web avec interactivité ;
- 1995, OAK devient Java, développement grâce au Web.
- Java devient généraliste. Grand succès dans milieux professionnels ;
- Langage propriétaire (Sun) mais licence ouverte ;
- Puces électroniques dédiées à Java ...

Une machine virtuelle

- Compilé et interprété ;
- Code source transformé en byte-code universel pour machine virtuelle (plugable sur navigateur web) ;
- Portabilité mais moins bonnes performances ;
- Développement de compilateurs à la volée (JIT - Just In Time).

Caractéristiques

- Syntaxe inspirée du C++, plus simple (Garbage Collector), POO plus propre ;
- Parallélisme : threads ;
- Distribué : Applets, RMI et Corba ;
- Pas de pointeur mais des références.

3.2 Types primitifs et structures de contrôle

- le type `boolean`, 2 valeurs `true` ou `false` sur 1 octet ;
- le type `char`, un caractère sur 2 octets ;
- les types `byte, short, int` et `long`, 4 types d'entiers stockés respectivement sur 1, 2, 4 et 8 octets ;
- les types `float` et `double`, 2 types de flottants stockés respectivement sur 4 et 8 octets ;
- déclarations n'importe où mais avant utilisation ;
- opérateurs usuels (C ou C++) ;
- les types primitifs sont passés par valeur lors des appels de méthode.

Structures de contrôle

- Les structures conditionnelles :
  - `if (cond) instruction1; [else instruction2;]`
  - `switch (selecteur) {`
    ```
    case c1 :  instructions1 ;
    ...
    case cn :  instructionsN ;
    default :  instructionsNP ;
    }
    ```
- Les structures répétitives ou boucles:
  - `for (initialisation; condition;`

`instructionDeSuite),`

  - `while (condition) instruction;`
  - `do instruction; while (condition).`

3.3 Classes et objets en Java

Une classe permet de définir un type d'objets associant des données et des opérations sur celles-ci appelées *méthodes*. Les données et les *méthodes* sont appelés *composants* de la classe.

Exemple : Robot se déplaçant sur une grille 2D

- données :
  - X, Y, orientation
- méthodes :
  - Initialisations
  - Avancer
  - TournerADroite


```
class Robot {
    // quelques constantes
    public static final int Nord = 1;
    public static final int Est = 2;
    public static final int Sud = 3;
    public static final int Ouest = 4;

    // données
    public int X; public int Y;
    public int orientation ;

    // constructeurs
    public Robot (int x, int y, int o)
    { X=x; Y=y; orientation=o; }
```

```
// autre constructeur
public Robot ()
{ X=0; Y=0; orientation=Nord; }

// methodes
public void avancer ()
{ switch (orientation)
  { case Nord : Y=Y+1; break;
    case Est : X=X+1; break;
    case Sud : Y=Y-1; break;
    case Ouest : X=X-1; break;
  };
}
```



```
public void tournerADroite ()
{ switch (orientation)
  { case Nord : orientation=Est; break;
    case Est : orientation=Sud; break;
    case Sud : orientation=Ouest; break;
    case Ouest : orientation=Nord; break;
  };
}
```

Déclaration, création et destruction d'objets

- Pour manipuler un objet, on déclare une *référence* sur la classe correspondant :

```
Robot totor;
```

- Pour créer l'objet et allouée une instance en mémoire, on utilise l'opération `new` :

```
totor = new Robot(5, 12, Sud);
```

- C'est un appel au *constructeur* défini par défaut si non préciser par le programmeur ;
- En Java, instanciation par *new* obligatoire sinon ils sont associés à `NULL`.

- `this` renvoie l'adresse de l'objet courant ;
- Destruction automatique des objets qui ne sont plus référencés par le *Garbage Collector*. On peut toutefois ajouter à chaque classe un service `finalize()`, qui sera appelé au moment de la destruction de l'objet :

```
class Robot { ...
   void finalize()
      {Systeme.out.println(``fin'');}
}
```

- Le nom d'un objet désigne une *référence*, c'est à dire une adresse. L'affectation de 2 objets de même nature recopie alors la référence.

```
UneClasse objet1 = new UneClasse();
UneClasse objet2 = objet1;
-> objet1 et objet2 même référence
```

Pour créer une copie, on utilise la méthode clone, prédéfinie dans chaque objet par défaut :

```
UneClasse objet3 = objet1.clone()
```

Illustration des références et des comparaisons :

```
totor = new Robot();
rotot = totor;
d2r2 = new Robot();
if (totor == d2r2) // faux
if (rotot == totor) // vrai
```


- Passage d'un objet en paramètres d'une fonction : on recopie la référence (la valeur de l'objet).
- On accède aux différents composants par la notation pointée :
  - `totor.X;`
  - `totor.avancer();`
- En Java, les implémentations des méthodes sont rédigées à l'intérieur de la définition de la classe.
  - boolean superieur (float x, float y) return x>y;

Tableaux

- Stocke des éléments de même type et dispose d'un index d'accès ;
- En Java, un tableau est un objet à part entière ;

```
double monTableau[];
```

ou encore

```
double[] monTableau;
```

- On vient de déclarer une référence, il faut maintenant le créer en mémoire avec la place nécessaire à tous ses élements :

```
monTableau = new double[10];
```


- Construction équivalente en une seule instruction :

```
double[] monTableau = new double[10];
```

- Initialisation :

```
double[] montableau = {0.0, 1.1, 3.5};
```

- On dispose d'opérations prédéfinies sur tous les tableaux. Par exemple, `length` renvoie la taille du tableau.

```
for (int i=0; i<montableau.length; i++)
    System.out.print(montableau[i]+" ");
```

- Mécanismes de vérification de dépassement de bornes des tableaux (par *exceptions*) que l'on décrira après. L'exception concernée se nomme `ArrayIndexOutOfBoundsException`.

Tableaux multidimensionnels

- Ce sont des tableaux de tableaux
- On peut les créer de taille homogènes ...
  `int[][] t = new int [5][10];`
- ... ou de tailles distincts

```
int m[][];
\\ création d'un tableau de 3 tableaux :
m = new int [3][] ;
\\ puis création séparée ces 3 tableaux :
m[0] = new int[2];
m[1] = new int[3];
m[2] = new int[1];
```


Composants de type `static`

- *composants*, données ou méthodes, non rattachés à des instances de la classe, mais qui sont communes à toutes. Pour cela il suffit de déclarer le composant avec le qualificatif `static`.
- Exemple :

```
class Robot  {    ...
  public static Robot
      leMeilleur(Robot r1, Robot r2) { ... }
}
```

- Utilisation de la méthode, préfixée du nom de la classe :

```
Robot.leMeilleur (totor, vigir)
```

Composants de type `public` et de type `private`

- Il s'agit ici du mécanisme d'encapsulation déjà développé dans le chapitre 2 (sur UML).
- On déclare `public`, les composants de l'objet accessibles en dehors de l'objet lui-même.
- On déclare `private`, les composants de l'objet qui ne sont accessibles que dans l'objet lui-même.
- On complètera ces notions de visibilité une première fois à propos des packages et une autre fois au sujet de l'héritage.

Chaînes de caractères

- Ce sont des objets, *instances* de la classe prédéfinie `String`, et elles référencent des chaînes constantes. On pourra les déclarer comme dans l'exemple suivant :

```
String ch1 = new String("bonjour");
```

  ou encore, sous une forme condensée qui est spécifique au type `String` :

```
String ch1 = "bonjour";
```

- La chaîne `"bonjour"` est ici constante mais `ch1` peut être réaffectée pour référencer une autre chaîne constante, comme dans l'exemple suivant :

```
String ch2 = "au revoir";
ch1 = ch2;
```

- On consultera la doc de l'API pour obtenir la liste des fonctions de manipulation. On présente ci-dessous les principales.
- `static String valueOf(int i)` renvoie une chaîne contenant la valeur de `i`. Existe pour tout type de paramètres primaires.
  `String.valueOf(12)` et qui retourne ici la chaîne "12".
- La méthode `boolean equals(String s)` compare le contenu de la chaîne courante avec la chaîne s.

- La méthode `String concat(String s)` renvoie la concaténation de la chaîne courante (celle qui va préfixée la fonction `concat`) et de `s`. Peut se faire aussi avec l'opérateur +.
- La méthode `int length()` renvoie la longueur de la chaîne courante.
- La méthode `int indexOf(int c)` renvoie la position de la première occurence du charactère de code ASCII `c`. Elle renvoie `-1` si ce caractère n'apparaît pas.
- La méthode `char charAt(int i)` renvoie le caractère à la position `i`.

Nous avons vu que les objets `String` référencent des chaînes constantes. `StringBuffer` permet de créer des instances sous forme de chaîne modifiable. Les différents constructeurs de la classe `StringBuffer` sont :

- `StringBuffer()` permettant de créer une chaîne vide ;
- `StringBuffer(int dim)` permettant de créer une chaîne de longueur `dim` ;
- `StringBuffer(String s)` permettant de créer une chaîne contenant `s`.

Les principales méthodes de la classe `StringBuffer` sont :

- `int length()` renvoyant la longueur de la chaîne ;

- `StringBuffer append (String s)` ajoutant `s` à la fin de la chaîne courante ;
- `String toString()` renvoyant dans une chaîne constante la chaîne modifiable courante.

3.4 Classes internes

Une classe interne est une classe définit à l'intérieur d'une autre classe. Il s'agit typiquement d'une classe locale qui peut être invisible à l'extérieur de la classe englobante avec le qualificatif private.

Exemple : classe pile avec des maillons chaînés

```
class PileEnt {
  private class Maillon {
    public int info;
    public Maillon suivant;
    public Maillon(int e, Maillon s)
```



```
      {info=e; suivant=s;}
  }
  private Maillon sommet;
  public PileEnt() {sommet=null;}
  public void empiler(int e)
    {sommet=new Maillon(e, sommet);}
  public void depiler()
    {sommet=sommet.suivant;}
  public int lire() {return sommet.info;}
  public boolean vide()
    {return (sommet==null); }
}
```

Pour accéder à une classe interne non privée, en dehors de la classe englobante, on devra utiliser la construction suivante :

`classeEnglobante.classeInterne`

Il y a pratiquement une classe interne spécifique par objet instancié de la classe englobante. La classe interne peut alors accéder aux composants de la classe englobante.

Exemple : On crée dans la classe `PileEnt` une classe interne `Parcours` pour pouvoir construire différents parcours utilisés simultanément.

```
class PileEnt {
  private int sommet;
  private int T[] = new int[100];
```



```
  public PileEnt()
    {sommet=0;}
  public void empiler(int e)
    {T[sommet++] = e;}
  public void depiler()
    { sommet-- ;}
  public int lire()
    { return T[sommet-1]; }
  public boolean vide()
    {return (sommet==0); }

  public class Parcours {
    private int courant;
    public Parcours()
```

```
      {courant=sommet;}
    public int element()
      {return T[courant-1];}
    public void suivant()
      {courant--;}
    public boolean estEnFin()
      {return (courant==0);}
  }
}
```

Exemple d’utilisation :

```
class TestPile {
  public static void main(String args[]) {
    PileEnt p= new PileEnt(); ?
```



```
    PileEnt.Parcours pa1= p.new Parcours();
    PileEnt.Parcours pa2= p.new Parcours();
    System.out.println(pa1.element());
    pa1.suivant();
    System.out.println(pa2.element());
    pa2.suivant();
    ...
  }
}
```

3.5 Organisation des fichiers sources d'un programme Java

- Programme Java = collection de classes dans un ou plusieurs fichiers sources dont l'extension est `"java"`. L'un de ces fichiers doit contenir une classe qui implémente la méthode `public static void main(String args[])`, comme cela est fait dans l'exemple précédent de construction de la classe PileEnt et de son programme de test.

Java, langage de développement objet

Commandes de compilation et de lancement d'un programme

Compilation des fichiers avec la commande `javac`. Par exemple, pour les deux fichiers relatifs à notre classe PileEnt et à son programme de test, on écrira :

```
javac PileEnt.java
javac TestPile.java
```

Deux fichiers : `PileEnt.class` et `TestPile.class` ont été générés et correspondent aux noms de toutes les classes définies dans les fichiers sources. Ce sont des fichiers en byte-code portables sur toute machine devant être traités par la machine virtuelle java lancée par la commande `java`. On

exécute donc le programme principal, qui est dans la classe `TestPile`, en tapant la commande :

```
java Testpile
```


Packages

- Un package regroupe un ensemble de classes sous un même espace de nomage.
- Les noms des packages suivent le schéma : name.subname
- lien entre package et répertoire :Une classe watch appartenant au package time.clock doit se trouver dans le fichier time/clock/watch.class
- L 'instruction package au début d'un fichier indique à quel package appartient ses classes
- En dehors du package, les noms des classes de ce package sont : packageName.className

- Les packages sont définis dans la variable d'environnement : `CLASSPATH`.
- `import packageName` permet d'utiliser des classes du package défini, sans avoir besoin de les préfixer par leur nom de package. On peut importer toutes les classes d'un package en utilisant un import du type `import packageName.*;`, mais,
- **ATTENTION**, l'import d'un niveau de package ne permet pas d'importer les packages qui sont en-dessous dans l'arborescence des répertoires.
- Voici un exemple d'illustration qui montre une organisation de classes java, dans différents répertoires et

leur utilisation.

- La variable d'environnement `CLASSPATH` doit être dans le fichier `.profile` ou dans `.bashrc`, par exemple, sous Unix, de la manière suivante :

  ```
  CLASSPATH = ~/myJavaClass
  ```

  Voici maintenant des extraits de différents fichiers rangés dans les répertoires indiqués :

  – le fichier /myJavaClass/bibAbstrait/pileUtil/PileEnt.java correspond à

  ```
  package bibAbstrait.pileUtil;
  public class PileEnt{ ... }
  ```

– le fichier /myJavaClass/bibAbstrait/fileUtil/FileEnt.java correspond à

```
package bibAbstrait.fileUtil;
public class FileEnt { ... }
```

– le fichier /myJavaClass/calUtil/TestFileEnt.java correspond à

```
package calUtil;
import bibAbstrait.fileUtil.*;
public class TestFileEnt {
  public static void main (String args[])
  {
    bibAbstrait.pileUtil.PileEnt x =
```


Java, langage de développement objet

```
      new bibAbstrait.pileUtil.PileEnt(3);
    FileEnt M = new FileEnt(3, 3);
    ...
  }
}
```

Utilisation des packages

Un package regroupe des classes qui portent sur un même domaine Au début de chaque fichier, on met package nomPackage La hiérarchie des package se retrouve au niveau de l'arborescence des fichiers et répertoires.

- Import nomPackage : pour utiliser une classe, on précisera le nom du package
- Import nomPackage.UneClasse : seule UneClasse est importée
- Import nomPackage.* : importe toutes les classes du package (mais pas les classes des sous-package !)

Visibilité des composants dans les packages

- Se fait grâce aux qualificatifs `private` ou `public`.
- En fait, elle est également liée aux localisations dans les packages. Ainsi, lorsqu'un composant ne précise pas sa nature (`private` ou `public`) alors il est, par défaut, `public` dans les classes du package (ou du répertoire) auquel il appartient et `private` en dehors.

Nous illustrons ces propos avec l'exemple suivant :

- Package P1 :

```
class C1 {
  public int xa;
```

```
    int xc;
    private int xd;
  }
  class C2 { ... }
```

- Package P2 :

```
  class C3 { ... }
```

Dans cet exemple, la classe `C2` peut accéder à `xa` et `xc`. Par contre `C3` ne peut accéder qu'à `xa` uniquement.

packages prédéfinis en Java

- `java.lang` qui correspond aux classes de base (chaînes, math, ...),
- `java.util` qui correspond aux structures de données (vector, piles, files, ...),
- `java.io` qui correspond aux entrées/sorties,
- `java.awt` qui correspond au graphisme et fenêtrage,
- `java.net` qui correspond aux communications Internet,
- `java.applet` qui correspond aux insertions dans des documents HTML.

Héritage
Construire une classe dérivée

- Permet de définir une classe dérivée à partir d'une autre classe dont elle *hérite* des composants et à laquelle on peut ajouter de nouveaux composants.
- Concept essentiel renforçant les propriétes de réutilisabilité des programmes objets.
- Hiérarchie et arbre d'héritage.

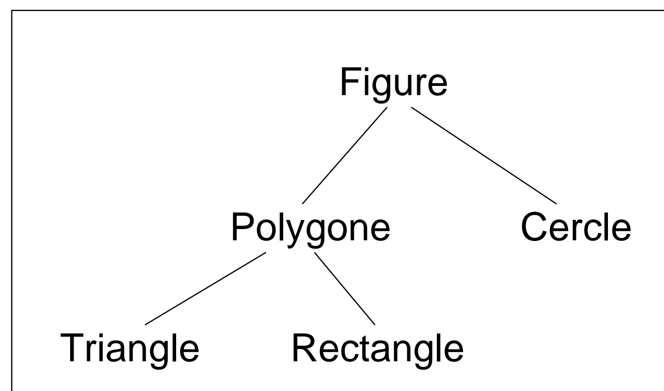

Figure 1: Arbre d'héritage de classes d'objets géométriques

- Pour définir une classe ClB qui dérive de la classe ClA, on fera une déclaration du type :

```
class ClB extends ClA { ... }
```

- Un objet de ClB est aussi un objet de la classe ClA,
- peut être utilisé partout où un objet de la classe ClA est attendu.
- On peut tester l'appartenance d'un objet à une classe grâce à l'opérateur `instanceof`, comme dans l'exemple qui suit :

```
ClB x;
if ( x instanceof ClA)
    System.out.println(
```


Java, langage de développement objet

```
        ``voici un objet ClA !'');
```

L'exécution des lignes précédentes provoque l'affichage de "`voici un objet ClA !`".

Constructeur d'une classe dérivée

- Il doit explicitement faire appel à un constructeur de la classe mère ;
- sinon appel implicite du constructeur par défaut (sans paramètre) de la classe mère qui doit exister (sinon erreur de compilation) ;

Appel explicite du constructeur de la classe mère :

- se fait par la méthode prédéfinie `super( ... )`,
- cet appel est obligatoirement la première instruction du constructeur.
- On ne peut donc transmettre que des valeurs de paramètres du constructeur courant lors de l'appel de `super( ... )`.

Accessibilité : public, protected et private

- Les composants d'une classe peuvent être qualifiés des termes :
  - public
  - private
  - protected : accessible dans les classes dérivées et les classes du même package.

Par exemple, soit la classe ClA définie de la manière suivante :

```
package aa;
public class ClA {
    protected int JJ;
    ...
```



```
}
```

et la classe ClB définie ainsi :

```
package bb;
import aa.*;
class ClB extends ClA {
    void PP() {
        JJ++;  // autorisé
        ClB b;
        b.JJ++ // autorisé
        ClA a;
        a.JJ++ // interdit
    }
}
```

Méthodes virtuelles et classes abstraites

- Une classe peut annoncer une méthode sans la définir, on dit alors que la classe est abstraite.
- Elle doit être introduite avec le mot clé `abstract`.
- Exemple, on définit la classe abstraite `ClA` suivante et une classe `ClB` qui en dérive.

```
abstract class ClA {
    abstract void fctP() ;
    void fctQ() { ... };
    ...
}
class ClB extends ClA {
```



```
    void fctP() { ... };
    ...
}
```

- En raison de sa définition incomplète, il est impossible d'*instancier* une classe abstraite
- ne sert qu'à la construction de classes dérivées : qui doivent redéfinir toutes les méthodes abstraites, pour ne pas l'être elles-mêmes et pouvoir ainsi être instanciées.

Un exemple : quelques objets géométriques

- Exemple d'implémentation des classes de la figure 1.
- Noter le constructeur de `Cercle` :
  - La classe `Figure` possède un composant de type `Point` pouvant être initialisé par le paramètre `x` du constructeur `Figure(Point p)`.
  - On définit la classe `Cercle` dérivant de `Figure`, dont le constructeur est défini par :

```
Cercle (Point centre, double r)
{ super(centre); ... }
```



```
class Point {
  double abscisse;
  double ordonnee;
  Point(double x, double y)
    {abscisse=x; ordonnee=y;}
  Point(Point p)
    {abscisse=p.abscisse;
     ordonnee=p.ordonnee;}
  static double distance(Point p, Point q) {
    double dx=p.abscisse-q.abscisse;
    double dy=p.ordonnee-q.ordonnee;
    return Math.sqrt(dx*dx+dy*dy);
  }
}
```

Nous définissons ensuite une classe abstraite pour définir le type `Figure` constitué de deux méthodes abstraites d'affichage et de calcul de périmètre :

```
abstract class Figure {
  private static final Point zero =
                         new Point(0,0);
        Point origine;
  Figure(){origine=zero;}
  Figure(Point p){origine=new Point(p);}
  abstract double perimetre();
  abstract void affiche();
}
```


La classe `Cercle` qui suit dérive de la classe `Figure` en implémentant ses deux méthodes abstraites `perimetre` et `affiche` :

```
class Cercle extends Figure {
  private static final double pi=3.141592;
  double rayon;
  Cercle(Point centre, double r)
    {super(centre); rayon=r;}
  double perimetre()
    {return 2*pi*rayon;}
  void affiche() {
    System.out.println("Cercle");
    System.out.println("rayon : " +
       rayon + " et centre : " +
```

```
            "(" + origine.abscisse + "," +
            origine.ordonnee + ")  ");
        }
    }
```

La classe `Polygone` dérive de la classe `Figure`. Elle se caractérise par un tableau de `Point` :

```
    class Polygone extends Figure {
      Point sommet[]= new Point[100];
      int nbs;
      Polygone(){nbs=0;}
      Polygone(Point[] m, int n) {
        super(m[0]);
        nbs=n;
```



```
        for (int i=0; i<n; i++)
          sommet[i]=m[i];
      }
      double lcote(int i) {
        if (i<nbs)
          return Point.distance(sommet[i-1],
                                sommet[i]);
        else
          return Point.distance(sommet[i-1],
                                sommet[0]);
      }
      double perimetre() {
        double somme=0;
        for (int i=1; i<=nbs; i++)
```

```
      somme += lcote(i);
    return somme;
  }
  void affiche() {
    System.out.println("Polygone");
    for (int i=0; i<nbs; i++)
      System.out.print(
          "(" + sommet[i].abscisse +
          "," + sommet[i].ordonnee + ")  ");
    System.out.println();
  }
}
```


La classe `Triangle` dérive de la classe `polygone` :

```
class Triangle extends Polygone {
  Triangle(Point[] m) { super(m,3); }
}
```

- La classe `Rectangle` dérive de la classe `Polygone`.
- Pour appeler le constructeur de `Polygone` qui possède le tableau de ses sommets en paramètres, il faudrait tout d'abord faire la construction de ce tableau à partir des caractéristiques de `Rectangle`, ce qui n'est pas possible, car l'appel de `super` doit être la première opération.
- Il faut donc utiliser le constructeur par défaut dans la classe `Polygone` qui sera appelé au début de l'exécution

du constructeur de `Rectangle`.

```
class Rectangle extends Polygone {
 double largeur;
 double longueur;
   Rectangle(Point m, double lo, double la) {
   // appel implicite du constructeur de
   // Polygone sans parametre
   // l'appel du constructeur de Polygone
   // avec parametres ne peut se faire car
   // il faut d'abord construire le tableau
   // a lui transmettre qui ne peut se faire
   // qu'apres l'appel explicite ou implicite
   // de super
         Point P1= new Point(m.abscisse+ lo,
```



```
                                    m.ordonnee);
         Point P2= new Point(m.abscisse,
                                 m.ordonnee+ la);
         Point P3= new Point(m.abscisse+ lo,
                                   m.ordonnee+ la);
         Point mr[]={m, P1, P3, P2};
         sommet=mr;
         nbs=4;
         largeur= la;
         longueur= lo;
      }
    }
```

Voici un programme principal de test :

```
class Geometrie {
  public static void main(String args[]) {
    Point P1= new Point(3,4);
    Point P2= new Point(4,4);
    Point P3= new Point(0,0);
    Point P4= new Point(1,0);
    Point P5= new Point(0,1);
    Point [] TabP={P3, P4, P5};
    Cercle c= new Cercle(P1,2);
    Rectangle r= new Rectangle(P2,5,2);
    Triangle t= new Triangle(TabP);
    Figure f; //autorise, mais pas new Figure !
    System.out.println("perimetre cercle");
```



```
    f=c; f.affiche();
    // appel de affiche de Figure
    // puis de sa forme derivee de cercle
    System.out.println("perimetre : " +
                              f.perimetre());

    f=r; f.affiche();
    System.out.println("perimetre : " +
                              f.perimetre());

    f=t; f.affiche();
    System.out.println("perimetre : " +
                               f.perimetre());
  }
```

```
    }
```

L'exécution du programme affiche :

```
java Geometrie
perimetre cercle
Cercle
rayon : 2.0 et centre : (3.0,4.0)
perimetre : 12.566368
Polygone
(4.0,4.0)  (9.0,4.0)  (9.0,6.0)  (4.0,6.0)
perimetre : 14.0
Polygone
(0.0,0.0)  (1.0,0.0)  (0.0,1.0)
perimetre : 3.414213562373095
```


Interfaces

- Une *interface* = un modèle de construction de classe dans lequel on n'indique uniquement que les en-têtes des méthodes.
- On dira qu'une classe *implémente* une *interface*, si elle redéfinit toutes les méthodes décrites dans cette *interface*.

Par exemple, on définit un modèle de problème par une interface qui comporte deux méthodes `poserProbleme` et `resoudreProbleme` :

```
interface AResoudre {
    void poserProbleme();
    void resoudreProbleme();
```

```
}
```

On peut alors construire une classe `equationPremierDegre` qui implémente l'interface `aResoudre` :

```
class EquationPremierDegre implements
                                AResoudre {
    double coefx, coef1, solution;
    void poserProbleme()
    {  // on lira coefx et coef1 }
    void resoudreProbleme()
    {  // on affecte une valeur a solution }
    ...
}
```


- Résoud les problèmes d'héritage multiple non autorisés (une classe ayant plusieurs classes mères) ;
- utilisation conjointe de l'hétitage et des interfaces.

Passage d'une fonction en paramètre d'une méthode

Le problème et le principe de la solution

- Pb : passage d'une fonction en paramètre d'une méthode.
- Exemple : méthode approchée de calcul d'une dérivée d'une fonction d'une variable réelle par un taux d'accroissement

$$f'(x) \simeq \frac{f(x+h/2) - f(x-h/2)}{h}$$

On veut que la fonction f reste "abstraite", c'est à dire paramètre de la méthode.

- En C ou C++, on passerait un pointeur, adresse du code de la fonction ... pas de pointeurs en Java.
- On créé une *enveloppe* de type objet contenant une méthode correspondante à l'évaluation de la fonction. C'est cette classe *enveloppe* qui correspond au paramètre à gérer.

L'implémentation de cette solution

- On définit une classe abstraite, ou mieux une interface, décrivant les fonctionnalités minimales de la classe correspondant au paramètre fonctionnel.

```
interface FoncD2D
   { public double calcul(double x); }
```

- Utilisation de cette interface pour décrire un procédé générique de calcul de dérivation numérique :

```
class DiffFinies {
    public double derivOrdre1
        ( FoncD2D f,
          double x,
          double h )
```



```
      {
        return (f.calcul(x+h/2) -
                f.calcul(x+h/2))/h ;
      }
}
```

- Pour utiliser ce procédé, il suffit maintenant de définir une fonction particulière dans une classe qui implémente l'interface `FoncD2D` :

```
class FoncCarre implements FoncD2D {
    public double calcul (double x)
       { return x*x ; }
}
```

- Le programme principal suivant va alors
  - construire un objet de la classe `FoncCarre`
  - construire un objet de la classe `DiffFinies` invoqué sur l'objet de type `FoncCarre`

```
class TestDiffFinies {
    public static void main (String args[]) {
        FoncCarre f = new FoncCarre();
        DiffFinies df = new DiffFinies();
        System.out.println
           ("Différences finies d'ordre un "+
            "en 1 de pas 0.01 : "+
            df.derivOrdre1(f, 1, 0.01));
    }
```



```
}
```

- Le résultat de l'exécution est :

```
java TestDiffFinies
Différences finies d'ordre un en 1
de pas 0.01 : 1.9999999999999685
```

Exceptions : notions générales

- La notion d'*exception* a pour but de simplifier le traitement de certaines situations, considérées exceptionnelles.
- Dans les langages sans gestion d'exception, nécessité d'utiliser de nombreuses instructions conditionnelles successives ... complexité du programme.
- En Java, on doit les gérer lorsque certains appels de méthodes sont susceptibles, de par leur conception, de déclancher des traitements d'exception.

Exceptions : implémentation

Une gestion d'exception est caractérisée par une séquence

`try - Catch - finally`

qui correspond typiquement au déroulement suivant :

```
try {
    //séquence succeptible de
    //déclancher une exception
    ...
}
catch (classException e1) {
    //traitement à effectuer
    //si e1 a été déclanchée
```

```
    ...
}
catch ( ....) { ... }
//autre déclanchement éventuel
finally {
    //traitement effectué avec ou
    //sans déclanchement d'exception
    ...
}
```


Exceptions : exemple

- Programme récupèrant les arguments fournis au lancement du programme.
- Il calcule et affiche la moyenne de ces arguments lorsque ce sont des entiers.
- Il déclanche un traitement d'exception si l'un des arguments ne correspond pas à un entier.

```
class exceptionCatch {
  static int moyenne (String[] liste) {
    int somme=0, entier, nbNotes=0, i;
    for (i=0; i<liste.length; i++)
      try {
```

```
        entier=Integer.parseInt(liste[i]);
        // conversion chaîne en valeur entière
        somme += entier; nbNotes++;
      }
      catch (NumberFormatException e) {
          System.out.println("note: "+(i+1)+
                                  " invalide");
      }
  return somme/nbNotes;
  }

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


Une exécution possible du programme est la suivante :

```
java exceptionCatch ha 15 12 13.5
note: 1 invalide
note: 4 invalide
moyenne 13
```

Définir sa propre exception

- Classe héritant de `Exception`.
- Possible redéfinition de `toString()` chaîne du message renvoyé à l'affichage de l'exception
- déclanchement par la clause `throws`, suivie de l'exception déclanchable
- Dans le corps de la méthode, on précisera dans quelle condition l'exception est déclanchée, en invoquant l'opérateur `throw`.

Définir une exception : exemple

Ccomplément du programme précédent déclanchant une exception, lors du calcul de la moyenne d'un tableau, s'il ne contient pas d'éléments.

```
class ExceptionRien extends Exception {
  public String toString() {
     return ("aucune note !");
  }
}
class ExceptionThrow {
  static int moyenne (String[] liste)
      throws ExceptionRien {
    int somme=0, entier, nbNotes=0, i;
```

```
    for (i=0; i<liste.length; i++)
       try {
         entier=Integer.parseInt(liste[i]);
         // conversion chaîne en valeur entière
         somme += entier; nbNotes++;
       }
       catch (NumberFormatException e) {
         System.out.println("note: "+(i+1)+
                             " invalide");
       }
       if (nbNotes == 0)
           throw new ExceptionRien();
     return somme/nbNotes;
  }
```



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

Voici deux exécutions successives du programme précédent :

```
java exceptionThrow q d 1 3 5.2
note: 1 invalide
```

```
note: 2 invalide
note: 5 invalide
moyenne 2

java exceptionThrow q d 3.1 a 5.2
note: 1 invalide
note: 2 invalide
note: 3 invalide
note: 4 invalide
note: 5 invalide
aucune note !
```


Entrées/Sorties
Classes de gestion de flux

Deux types d'entrées/sorties utilisables :

- Les entrées/sorties traditionnelles, c'est-à-dire utilisant les flux de communications “par défaut”, à savoir le clavier ou l'écran, ou encore les fichiers ;
- Les entrées/sorties basées sur des interactions avec un système de fenêtrage. Celles-ci seront développées dans le chapitre sur le graphisme.

Les principales classes de gestion de flux sont organisées suivant la hiérarchie d'héritage décrite dans la figure 2.

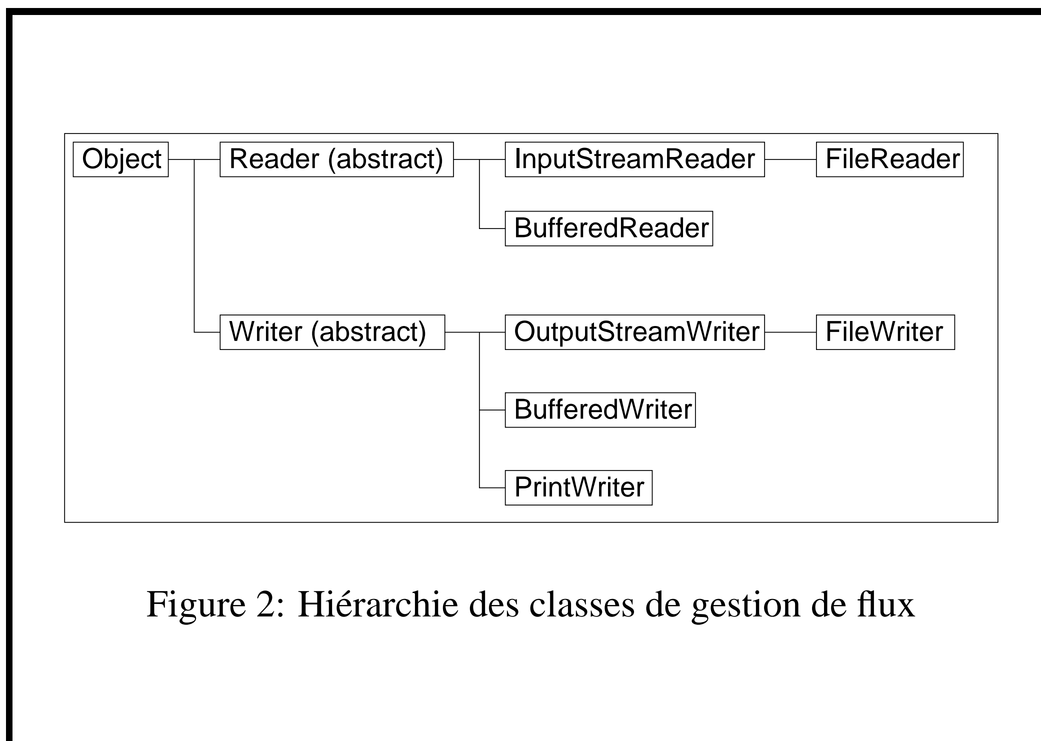

Figure 2: Hiérarchie des classes de gestion de flux

Dans cette hiérarchie, les classes suivantes apparaissent :

- La classe `Object` qui est la classe de base en Java, dont toutes les autres héritenté;
- Les classes abstraites `Reader` et `Writer` qui concernent respectivement les flux de caractères pour les lectures et les écritures ;
- Les classes `InputStreamReader` et `OutputStreamWriter` qui permettent de faire la traduction des données brutes en caractères UNICODE et inversement ;
- Les classes `BufferedReader` et `BufferedWriter` qui permettent l'utilisation d'une mémoire tampon pour

les entrées-sorties. Cette mémoire est indispensable pour l'utilisation des périphériques standards (écran et clavier) ;

- Les classes `FileReader` et `FileWriter` qui permettent l'utilisation de fichiers ;
- La classe `PrintWriter` qui permet l'écriture de données formattées semblables aux affichages à l'écran.

Cette liste n'est pas exhaustive mais correspond aux principales classes que nous serons amenés à utiliser dans la suite.

Saisies au clavier

Pour effectuer une saisie au clavier, on construit successivement :

- Un flux `InputStreamReader` avec le flux de l'entrée standard, à savoir `System.in` ;
- Un flux de lecture bufferisé de type `BufferedReader` à partir du flux précédent.

Exemple typique :

- lecture dans le flux bufferisé avec `readLine()` permettant la lecture d'un chaîne de caractères jusqu'à ce que l'on rencontre un saut de ligne.
- La chaîne de caractère est alors convertie en entier, avec `() parseInt`, ou en flottant, avec un construction plus complexe qui utilise `floatValue()`, sur l'objet de la classe `Float` obtenu en appelant `valueof()`.
- à partir du JDK 1.2, on pourra plus simplement appeler la méthode `parseFloat()`, similaire à `parseInt()`.

On remarquera que, dans cet exemple, il est nécessaire de gérer le déclanchement éventuel d'exceptions peuvant se produire,

- soit par un problème de lecture de flux qui provoque le déclanchement de l'exception `IOException` ;
- soit par un problème de conversion de chaîne de caractères en valeur numérique qui provoque l'exception `NumberFormatException`.

```
import java.io.*;
class Testio {
  public static void main(String[] args) {
    InputStreamReader fluxlu =
       new InputStreamReader(System.in);
    BufferedReader lecbuf =
       new BufferedReader(fluxlu);

    try {
      System.out.print(
         "taper 1 ligne de caracteres : ");
      String line = lecbuf.readLine();
      System.out.println("ligne lue : " + line);
```



```
      System.out.print(
         "taper 1 nombre entier : ");
      line = lecbuf.readLine();
      int i = Integer.parseInt(line);
      System.out.println("entier lu : " + i);

      System.out.print("taper 1 nombre reel : ");
      line = lecbuf.readLine();
      float f = Float.valueOf(line).floatValue();
      System.out.println("réel lu : " + f);

      System.out.println(
         "somme des deux nombres : " + (i+f) );
    }
```

```
    catch(IOException e) {
      System.out.println("erreur de lecture"); }

    catch(NumberFormatException e) {
      System.out.println(
         "erreur conversion chaine-entier"); }
  }
}
```


L'affichage obtenu à la suite de l'exécution du programme est le suivant :

```
java Testio
taper 1 ligne de caracteres : java sait lire
ligne lue : java sait lire
taper 1 nombre entier : 3
entier lu : 3
taper 1 nombre reel : 12.5
réel lu : 12.5
somme des deux nombres : 15.5
```

Lecture d'un fichier

Une lecture dans un fichier est effectué dans le programme suivant. Il est similaire au précédent avec, toutefois, quelques différences :

- `FileReader` est la classe instanciée pour construire le flux d'entrée à partir du nom du fichier. Cette instanciation pourra déclancher l'exception `FileNotFoundException`, si le fichier n'est pas trouvé.
- La méthode `close()` ferme le fichier.
- On utilise une boucle qui détecte la fin de fichier, suite au résultat d'une lecture qui renvoie la constante `null`.


```
import java.io.*;
class Testfile {
  public static void main(String[] args) {
    FileReader fichier=null;
    BufferedReader lecbuf;

    String line;
    float f, somme=0;
    int nbnombre=0;

    try {
      fichier = new FileReader("donnee.dat");
      lecbuf = new BufferedReader(fichier);
```

```
      while ( (line =
               lecbuf.readLine()) != null ) {
        f = Float.valueOf(line).floatValue();
        somme += f; nbnombre++;
        System.out.println("nombre lu : " + f);
      }
      if ( nbnombre > 0 )
        System.out.println("Moyenne : " +
                           (somme/nbnombre));
    }
    catch(FileNotFoundException e) {
      System.out.println(
           "fichier donnee.dat inexistant !"); }
```



```
    catch(IOException e) {
      System.out.println(
           "erreur de lecture"); }

    catch(NumberFormatException e) {
      System.out.println(
           "erreur conversion chaine-entier"); }

    finally {
      if (fichier!=null)
        try { fichier.close(); }
        catch(IOException e) {}
} } }
```

On exécute le programme précédent avec le fichier `"donnee.dat"` suivant :

```
2.3    1.2
3.4    2.1
5.2
```

L'affichage, produit par le programme, est alors le suivant :

```
java Testfile
nombre lu : 2.3
nombre lu : 1.2
nombre lu : 3.4
nombre lu : 2.1
nombre lu : 5.2
Moyenne : 2.84
```


Ecriture dans un fichier

- Le programme suivant va utiliser un fichier dans lequel on écrit.
- On construit un flux de type `FileWriter`, utilisé dans un tampon d'écriture (de type `BufferedWriter`).
- Un flux, de type `PrinterWriter`, permet alors d'effectuer des écritures formattées avec la méthode `println`.

```
import java.io.*;
class Testfileout {
  public static void main(String[] args)
        throws IOException {
```

```
    FileWriter fichier =
        new FileWriter("out.txt");
    BufferedWriter ecrbuf =
        new BufferedWriter(fichier);
    PrintWriter out =
        new PrintWriter(ecrbuf);
    out.println("coucou");
    out.println(5.6);
    System.out.println(
        "fin d'ecriture dans fichier out.txt");
    out.close();
  }
}
```


Compléments

- Lecture et écritures de données brutes avec les classes `DataInputStream` et `DataOutputStream`. Les fichiers, ainsi construits, ne sont pas lisibles directement sous un éditeur de texte, par exemple, mais leur taille est plus réduite.
- La classe `StringTokenizer` est une classe qui permet d'instancier un petit analyseur de texte qui peut, par exemple, découper des lignes en sous-chaînes de caractères, en reconnaissant un certain nombre de séparateurs (blancs, virgules, ...).

- La classe `java.io.File` permet de faire des manipulations de fichiers, similaires aux commandes d'un système d'exploitation. Elle permet par exemple, de lister un répertoire, de renommer ou supprimer un fichier, etc.