Java

Un tour d'horizon

Licence Info : L2

Damien Olivier

Université du Havre – LITIS

12 janvier 2023

Organisation du cours

1. Présentation de Java ;
2. Classes et objets en Java ;
3. Types et structures ;
4. Exceptions ;
5. Héritage ;
6. Le polymorphisme ;
7. La classe `Object` ;
8. Entrées/Sorties ;
9. Graphismes, fenétrages ;
10. AWT et Swing ;
11. Threads ;
12. Collections

Partie 1

Présentation de Java

Plan de la partie

Objectifs

- Applications indépendantes des machines/OS ;
- Applications structurellement distribuées sur réseaux ;
- Langage de programmation objets ;
  - Rien n'existe en dehors des classes ;
  - Même le `main` sera placé dans une classe.

Historique

- 1990/1993 : OAK langage pour domotique (télé interactive, ...) - SUN Microsystems. Codes petits, efficaces et indépendants de l'architecture. Développement : télé interactive – ... mais Web ++ !
- 1995 : OAK devient JAVA, popularisé par le Web (Java n'est pas Javascript).
- 1995- 1999 : Java $\Rightarrow$ langage généraliste. Succès considérable dépassant prévisions de SUN, adopté par tous les grands industriels de l'informatique.
- Langage propriétaire, mais licence ouverte.
- Puces électroniques Java, matériels embarqués . . .

Une machine virtuelle

- Langage compilé et interprété ;
- Code source transformé en code universel pour machine virtuelle ;
- Code universel exécuté par un interpréteur installé sur la machine ou dans une application (Navigateur Web) ;
- Portabilité mais moins bonne performance ;
- Aujourd'hui, compilateur à la volée en code machine (JIT).

Quelques propriétées importantes

- Syntaxe inspirée du C++, plus simple, POO plus propre ;
- Parallélisme : les Threads ;
- Distribué : Applets, RMI et Corba (1.2 / 2) ;

Pas de pointeurs mais des références.

Partie 2

Classes et Objets en Java

Plan de la partie

Définition d'une classe

- Une classe permet de définir un type d'objet associant données et méthodes ;
- Exemple : Robot se déplaçant sur une grille 2D

  données : `x, y, orientation`

  méthodes : `initialisations`
  `avancer`
  `tournerADroite`

Exemple

```
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 x1, int y1, int o)
 { x=x1; y=y1; orientation=o;}

 public Robot()
 { x=0; y=0; orientation=Nord; }
```

```
// méthodes
 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;
   }
 }
}
```

Définition d'une classe

- `public` : composant accessible partout
- `final` : composant constant
- `Robot` : désigne le(s) constructeur(s). Il est défini par défaut.

En résumé

classe = ?

- Définit le type et le comportement commun des instances ;

En résumé

classe = ?

- Définit le type et le comportement commun des instances ;
- Objet = instance d'une classe ;

En résumé

classe = attribut

- Définit le type et le comportement commun des instances ;
- Objet = instance d'une classe ;
- Attributs : variables d'instances, définissent l'état de l'objet ;

En résumé

classe = attributs + méthodes

- Définit le type et le comportement commun des instances ;
- Objet = instance d'une classe ;
- Attributs : variables d'instances, définissent l'état de l'objet ;
- Méthodes : opérations applicables à un objet de la classe ;

En résumé

classe = attributs + méthodes + instanciation

- Définit le type et le comportement commun des instances ;
- Objet = instance d'une classe ;
- Attributs : variables d'instances, définissent l'état de l'objet ;
- Méthodes : opérations applicables à un objet de la classe ;
- Instanciation : mécanisme de création d'un objet.

Déclaration et création d'objets

- Pour manipuler un objet, on déclare une référence sur la classe de cet objet :
  - `Robot totor;`
- Pour créer un objet, on instancie une classe en appliquant l'opérateur `new` sur un de ses constructeurs. Une nouvelle instance de cette classe est alors allouée en mémoire :
  - `totor = new Robot(5,12,Sud);`
- Toute classe possède un constructeur par défaut, implicite. Il peut être redéfini, éventuellement plusieurs fois comme en C++ .
- En Java, tous les objets doivent être instanciés par `new`, sinon ils sont associés à `null`.

Destruction d'un objet

- La destruction d'un objet est prise en charge par le garbage collector (GC) qui détruit les objets pour lesquels il n'existe plus de référence.
- Si la classe de l'objet possède la méthode `finalize`, celle-ci est appelée lorsque l'objet est détruit.

Exemple :

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

Utilisation des objets

Implémentation des méthodes

- Accès aux composantes par notation pointée
  - `totor.x;`
  - `totor.avancer();`
- Dans Java, les implémentations des méthodes sont toujours à l'intérieur de la définition d'une classe.
- Syntaxe du C / C++
  - `boolean superieur (float x, float y) { return x>y; }`
- L'instance ou objet courant est désigné par `this`

En résumé

objet = ?

En résumé

objet = identifiant

- identifiant : identificateur unique et invariant qui permet de référencer l'objet indépendemment des autres objets ;

En résumé

objet = identifiant + comportement

- identifiant : identificateur unique et invariant qui permet de référencer l'objet indépendemment des autres objets ;
- Comportement : défini par les méthodes applicables à sa classe ;

En résumé

objet = identifiant + comportement + état

- identifiant : identificateur unique et invariant qui permet de référencer l'objet indépendemment des autres objets ;
- Comportement : défini par les méthodes applicables à sa classe ;
- Etat : valeur simple ou structurée de l'objet.

Composants non liés à des objets : static

- Une fonction non rattachée à une instance d'une classe, commune à toutes les instances, sera qualifiée de `static`
- Exemple : on ajoute une fonction `leMeilleur` à la classe Robot non rattachée à une instance

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

Composants non liés à des objets : static

- Pour appeler une telle procédure, on la préfixe par le nom de la classe et non par le nom dfinal'une instance.
  `Robot.leMeilleur(totor, vigir)`
- Une méthode `static` n'a pas d'instance de rattachement : `this` ne peut pas y être utilisé.
- Rappel : une fonction n'existe pas seule, elle est toujours rattachée à une classe.

Abstraction de la représentation : private et public

- Comme en C++, on doit utiliser avec pertinence la description d'une interface publique et masquer tous les composants internes avec le qualificatif `private`.
- Exemple : construction d'une pile d'entiers

```
class PileEnt {
  private int sommet;
  private int tab[] = new int[100];
  public PileEnt() {sommet=0;}
  public void empiler(int e) {tab[sommet++] = e;}
  public void depiler() { sommet-- ;}
  public int lire() { return tab[sommet - 1]; }
  public boolean vide() {return (sommet==0); }
}
```

Classe interne

- Classe définie à l'intérieur d'une autre classe ;
- Classe locale, elle 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); }
}
```

Classe interne

- Pour accéder à une classe interne non privée, en dehors de la classe englobante : `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

Classe interne

```
class PileEnt {
  private int sommet;
  private int tab[] = new int[100];
  public PileEnt() {sommet=0;}
  public void empiler(int e)
    {tab[sommet++] = e;}
  public void depiler() { sommet-- ;}
  public int lire()
    { return tab[sommet-1]; }
  public boolean vide()
    {return (sommet==0);}

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

Exemple d'utilisation :

```
public 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();
  }
}
```

On pourra remarquer que l'on accède au niveau de la classe `Parcours` à l'attribut `tab` de la classe `PileEnt` qui est privé.

Structures des fichiers sources

- Programme Java = collection de classes dans 1 ou plusieurs fichiers sources d'extension : `.java`
- Le compilateur se lance par la commande :
  `javac fichier.java`

$\Rightarrow$ Il génère 1 fichier par classe, définie dans le programme source, portant le nom de la classe et l'extension `.class`

- L'exécution s'effectue par `java Testpile` où `Testpile.class` correspond à une classe contenant une méthode de nom `main`.

Packages et fichiers

- 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`

Packages

- Les répertoires où Java effectue sa recherche de packages sont définis dans la variable d'environnement : `CLASSPATH`
- L'instruction `import` packageName permet d'utiliser des classes sans les préfixer par leur nom de package.
- Java propose des bibliothèques prédéfinies regroupées par thèmes :
  - java.lang : classes de base (chaînes, math, ...)
  - java.util : structures de données (vecteurs, piles, files, ...)
  - java.io : entrées/sorties
  - java.awt : graphisme et fenêtrage
  - java.net : communications Internet
  - java.applet : insertion dans pages HTML

Packages

- CLASSPATH = $JAVA_HOME/lib/classes.zip;$HOME/classes
- fichier ~/classes/graph/2D/Circle.java :

```
package graph.2D;
public class Circle { ... }
```

- fichier ~/classes/graph/3D/Sphere.java :

```
package graph.3D;
public class Sphere {... }
```

- fichier ~/classes/dessin/Mainclass.java :

```
package dessin;
import graph.2D.*;
public class Mainclass
{ public static void main(String args[])
  {
   graph.3D.Sphere s1= new graph.3D.Sphere(100);
   Circle c2= new Circle(70);
  }
}
```

Utilisation d'un package

- Un package regroupe des classes qui portent sur un même domaine (p.e une structure de données) ;
- Au début de chaque fichier on met `package nomPackage;` ;
- La hiérarchie d'un package se retrouve au niveau de l'arborescence des fichiers et répertoires ;
-

Exemple	*Role*
`import nomPackage;`	Les classes ne peuvent pas être utilisées simplement par leur nom et il faut préciser le nom des packages
`import nomPackage.*;`	Toutes les classes du package sont importées
`import nomPackage.UneClasse;`	Seule `UneClasse` est importée

- Attention `import nomPackage.*;` n'importe pas les sous packages.

Encapsulation des membres

- Les accessibilités aux composants se font par les qualificatifs : `public` et `private`. Elles sont liées aussi aux localisations dans les packages.
- Lorsqu'un composant ne précise pas sa nature (privée ou publique), il est public dans le package et privé en dehors.

- Exemple :

```
package P1;
class C1 {
   public int a;
   int b;
   private int d;
}
class C2  { ... }
package P2;
class C3 { ... }
```

Encapsulation des membres

- Les accessibilités aux composants se font par les qualificatifs : `public` et `private`. Elles sont liées aussi aux localisations dans les packages.
- Lorsqu'un composant ne précise pas sa nature (privée ou publique), il est public dans le package et privé en dehors.

- Exemple :

```
package P1;
class C1 {
   public int a;
   int b;
   private int d;
}
class C2  { ... }
package P2;
class C3 { ... }
```

- C2 peut accéder à `a` et `b` ;
  C3 ne peut accéder qu'à `a` uniquement.

Utilisation

- On utilise le type `enum` qui permet de déclarer des constantes ;
  - `enum Feu {ROUGE, ORANGE, VERT}`
- Déclaration d'une variable `Feu tricolore = Feu.ROUGE;` ;
- Les énumérations héritent de la classe `java.lang.Enum` ;
- Les énumérations ne peuvent pas implémenter des interfaces ;
- Des méthodes et les variables peuvent être déclarées comme dans une classe ;
- Les variables et les constructeurs sont déclarés private ;
- Les énumérations sont utilisées avec l'instruction Switch.

Exemple EnumEssai.java

Partie 3

Types et Structures du Langage

Plan de la partie

Types primitifs

- Ce ne sont pas des classes
- `boolean` (true/false), `byte` (1 octet), `char` (2 octets), `short` (2 octets), `int` (4 octets), `long` (8 octets), `float` (4 octets), `double` (8 octets)
- Comme en C++, on peut les déclarer n'importe où
- Opérateurs usuels identiques à ceux du C (Attention : les affectations entre types différents doivent être "castées")

Les types primitifs sont passés par valeur

Structures du langage

- Essentiellement les mêmes qu'en C
  `=, if, switch, for, while, do while`
- Attention :
  l'affectation entre 2 objets recopie une référence

  de même, une comparaison (==) entre 2 objets compare leurs références et non leurs contenus

  - `UneClasse objet1 = new UneClasse();`
    `UneClasse objet2 = objet1;`
    `objet1` et `objet2` contiennent la même référence et désignent donc le même objet ;
    Pour créer une copie on utilise la méthode `clone()` ;
    `UneClasse obj1 = new UneClasse();`
    `UneClasse obj2 = obj1.clone();`
  - Références et comparaison d'objets

```
totor = new Robot(0, 0, Nord);
rototo = totor;
d2r2  = new Robot(0, 0, Nord);
if(totor == d2r2) // Faux .....
if (rototo == totor) // Vrai ....
```

Les tableaux

- Comme les instances de classes, ce sont des références ;
- Déclaration : `Robot r[]; //ou Robot[] r;` ;
- Création : `r= new Robot[20];` ;
- En 1 opération : `Robot[] r = new Robot[20];` ;
- Initialisation :
  `int[] premiers = {1, 2, 3, 5, 7, 7+4};` ;
- On peut accéder à la taille du tableau : `r.length` ;
- Vérification des dépassements des bornes (`ArrayIndexOutOfBoundsException`) ;

Les tableaux multidimensionnels

- Ce sont des tableaux de tableaux
- On peut les créer de tailles homogènes :

```
int[][] t= new int[5][10];
```

- ou de tailles distinctes

```
int m[][];
m = new int [3][]; //tableau de 3 tableaux d'entiers
m[0] = new int[2];
m[1] = new int[3];
m[2] = new int[1];
```

La classe String

- Ses instances sont des objets référençant des chaînes constantes

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

- La chaîne "bonjour" est constante mais ch1 peut être réaffectée pour référencer une autre chaîne constante
  `String ch2 = "au revoir"; ch1 = ch2;`
- Quelques méthodes :
  `static String valueOf(int i)` : chaîne représentant l'entier `i`
  `static String valueOf(t x)` : chaîne représentant `x` de type `t`

La classe String

Quelques méthodes (suite)

- Les méthodes précédentes non liées à des objets (`static`) sont utilisées en les préfixant par le nom de la classe.
- `String.valueOf(12)` est évaluée en "12"
- `boolean equals (String s)` : compare le contenu de `this` et `s`
- `String concat (String s)` : renvoie la concaténation de `this` et `s`
  Remarque : la concaténation peut aussi se faire avec l'opérateur +
- `int length()` : renvoie la longueur de `this`
- `int indexOf (int c)` : renvoie la position de la première occurrence du caractère de code ASCII `c` (ou -1 si absent)
- `char charAt (int i)` : renvoie le caractère à la position `i`

La classe StringBuffer

- Ses instances sont des objets référençant des chaînes modifiables
- Constructeurs :
  `StringBuffer()` : chaîne vide
  `StringBuffer(int length)` : chaîne de taille `length`
  `StringBuffer(String s)` : chaîne contenant `s`
- Quelques méthodes :
  `StringBuffer append (String s)` : ajoute `s` en fin de `this`
  `String toString()` : renvoie la chaîne constante contenant `this`
  `int length()` ...

Entrées/Sorties utilisant des chaînes

```
package EntreeSortie;
import java.io.*;
public class Console
{
 public static void printMessage(String message)
 {
  System.out.print(message + " ");
  System.out.flush();
 }

 public static String readChaine()
 {
  int carac;
  String resultat = "";
  boolean lu = false;
  while (!lu)
  {try
    {
     caractere = System.in.read();
     if (carac < 0 || (char)carac == '\n')
       lu = true;
       else
         if ((char)caractere != '\r')
            resultat = resultat + (char)carac;
     }
     catch(java.io.IOException e) {lu = true;}
  }
  return resultat;
 }

 public static String readChaine(String message)
 {
  printMessage(message);
  return readChaine();
 }
```

```
 public static int readInt(String message)
 {
  while(true)
  {
   printMessage(message);
   try
    {
     return Integer.valueOf
                  (readChaine().trim()).intValue();
    }
    catch(NumberFormatException e)
    {
     System.out.println("Ce n'est pas un entier !");
    }
  }
 }

 public static double readDouble(String message)
 {
  while(true)
  {
   printMessage(message);
   try
    {
     return Double.parseDouble(readChaine().trim());
    }
    catch(NumberFormatException e)
    {
     System.out.println("Ce n'est pas un réel !");
    }
  }
 }
}
```

Entrées/Sorties en Java 1.5 (Tiger)

Il existe deux nouvelles classes : `Formatter` et `Scanner` ;

- `Formatter` qui permet de réaliser un formatage des sorties :

```
import java.util.*;
public class TestFormatter {
  public static void main(String[] args) {
   Formatter formateur = new Formatter();      // On cré
    e un objet formateur
   formateur.format("Résultat = \4.3f \n ", 7./3);  //
    4 chiffres, 3 après la ,
   String resultat = formateur.toString();
   System.out.println("--" + resultat + "--");
  }
}
```

La classe Scanner

- Permet la lecture dans un flux ;
- La méthode `next()` renvoie ce qui a été lu sous la forme d'une `String` ;
- La méthode `close()` ferme le flux.
- Exemple :

```
import java.util.*;
public class TestScanner {
  public static void main(String[] args) {
  Scanner entree = new Scanner(System.in);
  String chaine = entree.next();
  System.out.println(chaine);
  //entree.close(); On ne ferme pas System.in
  }//        c'est la machine virtuelle qui le fait
}
```

- On peut lire des types primitifs avec `nextXXX` avec `XXX` type primitif.

```
import java.util.*;
public class TestScannerEntier {
  public static void main(String[] args) {
  Scanner entree = new Scanner(System.in);
  int entier = entree.nextInt();
  System.out.println(entier);
  // entree.close();
  }
}
```

Partie 4

Traitement des exceptions

Plan de la partie

Exceptions

- Traitement des cas exceptionnels (p.e. cas d'erreurs)
- Sans gestion particulière (en C, par exemple), insertion d'instructions conditionnelles
- $\Rightarrow$ alourdit le code
- $\Rightarrow$ logique du traitement "normal" noyé dans les traitements conditionnels
- Gestion particulière des exceptions :
  Existe en C++, pas d'obligations d'utilisation
  En Java, obligation de gérer le traitement des exceptions lors de l'appel d'une méthode pouvant en déclencher.

try - catch - finally

```
try
{
  // séquence susceptible de déclencher une exception
} catch (ClasseException e1)
  { //traitement à effectuer si exception déclenchée\\
  }
  catch ( ... ) { ... } // Autre exception
  finally { ... } // traitement à faire avec ou sans déclenchement d'exception
```

try - catch - finally

- Quand l'exception se déclenche :
  - Sortie définitive du bloc `try` ;
  - Exécution des instructions du catch correspondant ;
  - Éventuellement exécution du `finally` ;
  - Poursuite du programme après bloc `try-catch-finally` ;
- Déclenchement d'une exception non gérée par un `catch` $\Rightarrow$ arrêt du programme.
- Si une méthode ne prévoit pas de bloc `catch` pour lever une exception susceptible d'être déclenchée, introduire une clause `throws` dans son en-tête

```
public void methode() throws ExceptionNonLevee { ... }
```

Exemple d'exception levée

```
public 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
     ]);
       // convertion 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));}
}
```

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

Exceptions multiples

- Avant Java 1.7

```
import java.util.Scanner;
public class PlusieursExceptions
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (n%47  ==  0)
                System.out.println(n + " est divisible par 47");
        }
        catch (ArithmeticException ex)
        {
            System.err.println("Pb arithmétique " + ex);
        }
        catch (NumberFormatException ex)
        {
            System.err.println("Problème de format " + ex);
        }
    }
}
```

Exceptions multiples

- Depuis Java 1.7

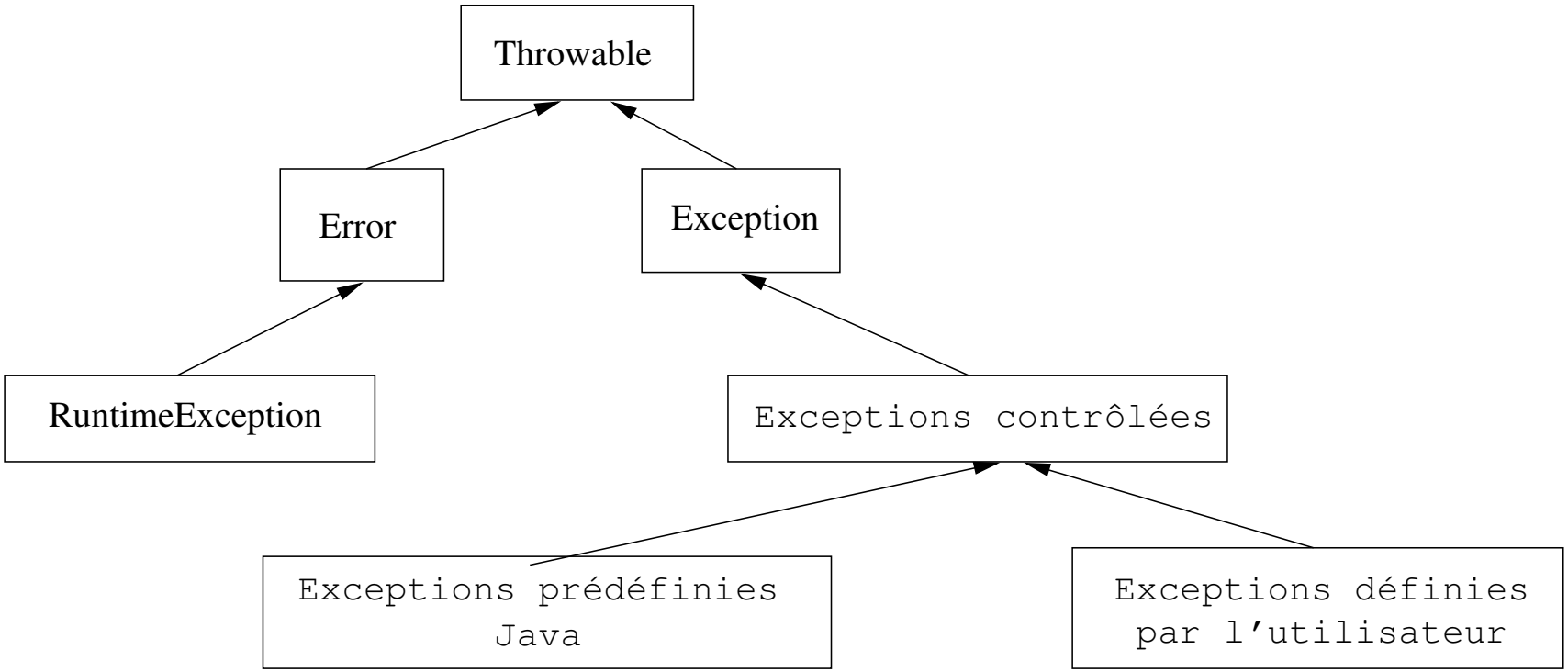
```
import java.util.Scanner;
public class PlusieursExceptions
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (n%47  ==  0)
                System.out.println(n + " est divisible par 47");
        }
        catch (ArithmeticException | NumberFormatException ex )
        {
            System.err.println("Exception  " + ex);
        }
    }
}
```

Exemple d'exception non levée

```
package EntreeSortie;
import java.io.*;
public class Console
{
  // ....

 public static String readChaine()
   throws IOException
 {
  int caractere;
  String resultat = "";
  boolean lu = false;
  while (!lu)
    {
     caractere = System.in.read();
     if (carac < 0 || (char)carac == '\n')
       lu = true;
       else
         if ((char)caractere != '\r')
            resultat = resultat + (char)carac;
     }
  return resultat;
 }
}
```

Arbre d'héritage des exceptions



Définir sa propre exception

```
class ExceptionRien extends Exception {
// classe dérivant de la classe Exception
public ExceptionRien(String s) {
      super(s);
   }
}

public class ExceptionThrow {
   static int moyenne(String[] liste) throws
      ExceptionRien {
   // indique que l'exception risque d'être déclenchée
   int somme = 0, entier, nbNotes = 0, i;
      for (i = 0; i < liste.length; i++) {
         try {
            entier = Integer.parseInt(liste[i]);
            // convertion chaîne en valeur entière
            somme += entier;
            nbNotes++;
         } catch (NumberFormatException e) {
            System.out.println("note:" + (i + 1) + "
      invalide");
         }
      }
      if (nbNotes == 0)
         throw new ExceptionRien("Aucune note n'est
      valide");
      return somme / nbNotes;
   }

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

Pour définir sa propre exception

- Définir une classe héritant de `Exception` ;
  - Cette notion sera précisée dans le chapitre suivant ;
- On aurait pu écrire `catch(Exception e)` car un objet de type `NumberFormatException` ou de type `ExceptionRien` est également une `Exception` par héritage.

Utilisation de la classe `Throwable`

- Classe mère de toutes les exceptions ;
- Descendante directe de la classe `Object` ;
- Possède en particulier les méthodes suivantes :

Méthodes	*Rôle*
`Throwable()`	Constructeur
`Throwable(String)`	La chaine définit l'exception pouvant être lue dans un `catch`
`getMessage()`	Lecture du message
`printStackTrace()`	Affiche l'exception et l'état de la pile d'exécution

Exemple

```
public class UtilisatThrowable {
 public static void main(String argv[])
 {
  try{
      System.out.println("Essai division par 0 = " + 3/0);
     } catch(ArithmeticException exception)
       {
        System.out.println("Message  : " + exception.getMessage());
        System.out.println("toString : " + exception.toString());
        System.out.print("Pile     : ");
  exception.printStackTrace();
  }
  }
}
```

```
Message  : / by zero
toString : java.lang.ArithmeticException: / by zero
Pile     : java.lang.ArithmeticException: / by zero
        at UtilisatThrowable.main(UtilisatThrowable.java:7)
```

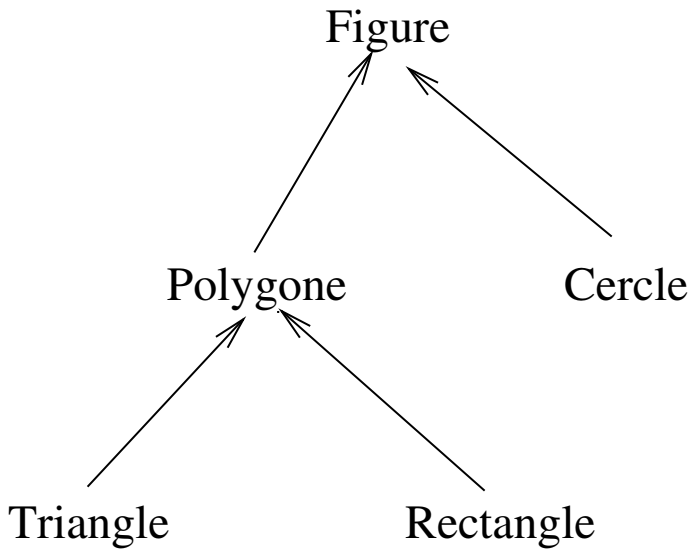
Partie 5

Héritage en Java

Plan de la partie

Qu'est ce que l'héritage ?

- Permet de définir une classe à partir d'une autre classe dont elle hérite des composants et à laquelle on ajoute de nouveaux composants ;
- Notion de hiérarchie de classes que l'on peut représenter par un arbre d'héritage ;
- Héritage simple en Java + notion d'interfaces.



Construire une classe dérivée

```
class Personne
{ String nom;
  int nbEnfants;

  Personne (String n)
   {nom=n; nbEnfants=0;}
}

class Salarie extends Personne
{ int salaire;
  int prime()
  {
   return 5*salaire*nbEnfants/100;
  }
  Salarie (String n, int s)
  {
   super(n);
   salaire=s;
  }
}
```

- Appel du constructeur de la classe de base avec la fonction prédéfinie `super()` ;
- Un objet de classe `Salarie` est également de classe `Personne` : il peut être utilisé partout où une instance de `Personne` est attendue ;
- On peut tester l'appartenance d'un objet à une classe par `paul instanceof Salarie` qui vaut `true` si paul est un objet de la classe `Salarie`.

Constructeur d'une classe dérivée

- Doit explicitement faire appel à un constructeur de la classe mère ...
  ... sinon appel implicite du constructeur par défaut (sans argument) de la classe mère qui doit exister (sinon erreur de compilation)
- L'appel explicite du constructeur de la classe mère se fait avec `super(...)` : c'est obligatoirement la première instruction $\Rightarrow$ on ne peut donc rien faire avant et on ne peut donc que transmettre des valeurs d'arguments à `super(...)`
- Exemple : ConstructeurEtHeritage.java et sa trace.

public, protected et private

Les composants d'une classe possède éventuellement un qualificatif d'accessibilité

- `public` : accessible en dehors de la classe ;
- `private` : inaccessible en dehors de la classe ;
- `protected` : accessible dans toutes les classes dérivées et les classes du même package ; classes du même package qui n'en dérivent pas ;
- aucun attribut : accessible dans les classes qui font partie du même package.

```
package aa;
public class A
{...
 protected int jj;
... }

package bb;
import aa.*;
class B extends A
{ ...
  void pp()
  {  jj++;    //autorisé
     B b = new B();
     b.jj++; //autorisé
     A a = new A();
     a.jj++; //interdit
  }
}
```

public, protected et private

Pour résumer

Modificateur d'accès	Classe	Package	Sous-classe	Reste du monde
`public`	O	O	O	O
`protected`	O	O	O	N
Pas de modificateur	O	O	N	N
`private`	O	N	N	N

Empécher l'héritage

- Pour interdire la dérivation possible d'une classe donnée, on la qualifie de `final`
  ```
  final class Point  //...
  ```
- Une méthode peut être `final`, aucune classe dérivée peut la redéfinir.

Pourquoi utiliser `final` ?

- *securité* Lorsqu'un objet d'une classe dérivée utilise une méthode définie dans une classe mère, on ne sait pas si il s'agit d'une méthode qui a été redéfinie sauf si la méthode a été déclarée `final` dans la classe mère.
- *efficacité* Le compilateur peut remplacer les méthodes `final` par du code en ligne (ressemble aux fonctions inline C++)

Méthodes virtuelles et classes abstraites

Une classe peut annoncer une méthode sans la définir : on dit que la classe est abstraite. Elle doit être introduite avec le mot clé `abstract`

```
abstract class A
{ ...
  abstract void p();
  void q() { ... }
}
```

```
class B extends A
{ ...
void p() { ... }
void q() { ... }
}
```

Impossible d'instancier une classe abstraite.

Ne sert que pour la construction de classe dérivée.

Interface

- Description d'une classe complétement abstraite le plus souvent.
- Abstraction pure qui permet un découplage maximal entre un service et son implémentation
  - Les méthodes sont toutes abstraites (pas de corps), sauf celles par défaut et les statiques (cf. suite) ;
  - La classe n'a pas d'attribut de classe sauf statique ;
  - Les méthodes sont publiques.

```
interface Pile {
  boolean estVide();
  void empiler(Object o);
  Object sommet();
  Object depiler();
}
```

- Une interface définit un type d'objet $\Rightarrow$ on peut définir une variable `Pile maPile;` ;
- Une interface est utile pour définir des types d'objets transversalement à la hierarchie des classes.

Utilisation

- Une classe peut fournir l'implantation d'une ou plusieurs interfaces ;
- La classe définit chaque méthodes des interfaces qu'elle implante ;
- Exemple :

```
class PileParTableau
  implements Pile
{
  private Object[] contenu;
  private int taille;
  PileParTableau(int max) {
    contenu = new Object[max];
  }
  public boolean estVide() {
    return taille == 0;
  }
  public void empiler(Object o) {
    contenu[taille] = o;
    taille++;
  }
  public Object sommet() {
    return contenu[taille - 1];
  }
  public Object depiler() {
    taille--;
    return contenu[taille];
  }
}
```

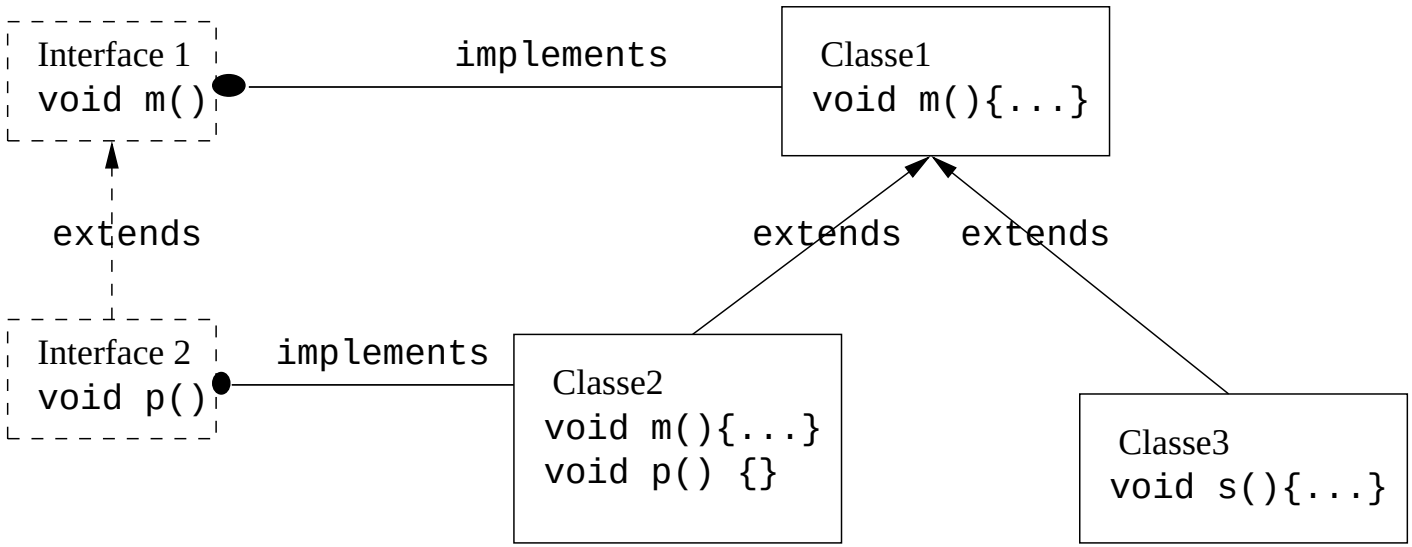
Utiliser une interface comme marqueur

- Une interface de marqueur en Java est une interface vide qui n'a ni champs ni méthodes.
- Elle fournit à l'exécution une information.
- Il existe de base trois interfaces : `java.lang.Cloneable`, `java.io.Serializable`, `java.rmi.Remote`.

Le cas Cloneable

- Chaque classe hérite de la classe `Object` et possède donc une méthode `clone()` ;
- Cette méthode n'est pas toujours correctement définie (cf. suite) ;
- En implémentant l'inteface vide `java.lang.Cloneable` vous indiquez que l'utilisation de la méthode `clone()` est correcte ;
- Autrement Java lancera une exception `CloneNotSupportedException` qu'il faudra traiter.

Héritage et interface



Exemple développé :

- Les interfaces : `Interface1.java` et `Interface2.java`
- Les classes : `Classe1.java`, `Classe2.java`, `Classe3.java` et `MainClasse.java`
- La trace : `HeritageEtInterface.txt`

Héritage et interface

- A quoi sert une interface ?
  - Garantir aux utilisateurs d'une classe que ses instances possèdent certaines propriétés (ex : comparable) ou peuvent assurer certains services ;
  - Polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage.
- Forme d'héritage multiple
  - Il faut distinguer l'héritage de classe (i.e. de code) de l'héritage d'interface (i.e. du type et du comportement attendu de l'objet) ;
  - Exemple : la classe `Hero` hérite de `Personnage` et implémente les interfaces `PeuVoler` et `PeuNager` dont la trace est la suivante ...

Méthodes statiques

- Une méthode statique est un élément que l'on peut rencontrer dans une classe normale ;
- L'appel statique se fait au travers de la classe

Méthodes par défaut

- Depuis Java 8, il est possible d'avoir une implémentation par défaut ;
- Il suffit de fournir un corps à la méthode, et de la qualifier avec le mot-clé `default` ;
- Les classes filles sont alors libérées de l'obligation de fournir elles-mêmes une implémentation à cette méthode - en cas d'absence d'implémentation spécifique, c'est celle par défaut qui est utilisée.

```
public interface ParDefaut {
    public void sansImplementation();
    public default void parDefaut1() {
        System.out.println("Interface -> parDefaut1()");
    }
    public default void parDefaut2() {
      System.out.println("Interface -> parDefaut2()");
    }
}

public class ImpInterface implements ParDefaut {
  public void sansImplementation() {
    System.out.println("ImpInterface -> sansImplementation
        ()")
  }
  public void parDefaut1() {
    System.out.println("ImpInterface -> parDefaut1()")
  }
}

public class Demo {
  public static void main(String[] args) {
    ImpInterface exemple = nex ImpInterface();
    exemple.sansImplementation();
    exemple.parDefaut1();
    exemple.parDefaut2();
  }
}
```

Signature et polymorphisme

- Une classe `ClasseB` hérite de `ClasseA` et `uneMethode` est définie dans `ClasseA` et redéfinie dans `ClasseB` ;
- Ceci est possible car les signatures sont différentes ;
- La signature comprend le nom de la classe, le nom de la méthode et le type des paramètres ;
- Cela permet de mettre en œuvre le polymorphisme.

Définition

- Capacité pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel est adressé le message ;
- Quelle méthode est exécutée ?

```
ClasseA a = new ClasseB(5);
        // a est un objet de ClasseB
  // mais déclaré de ClasseA
a.uneMethode();
```

- La méthode appelée ne dépend que du type réel de l'objet a c'est donc la méthode de la `ClasseB` qui est effectuée.
- Exemple et sa trace.

Liaison retardée

- Soit `Hero` la classe réelle de `donald` à qui on envoie le message `donald.cri()`
- Le code de la classe `Hero` contient la définition de la méthode `cri()`, elle est exécutée.
- Autrement la recherche s'effectue dans la classe mère et ainsi de suite jusqu'à trouver une définition.
  Exemple : code et trace.

Conversion - cast

Il faut distinguer :

- Le type réel d'un objet qui est déterminé par le constructeur qui a créé l'objet ;
- Le type déclaré.

Le cast est une conversion

- On convertit un objet en un type qui n'est pas le type déclaré ou réel de l'objet ;
- Uniquement des casts entre classes appartenant à une même branche de l'arbre d'héritage.

Cast

Le cast peut être :

- Conversion classe fille → classe mère (implicite eventuellement) ;
  - Souvent utilisé pour le polymorphisme ;
  - Exemple : code et trace.
- Conversion classe mère → classe fille (explicite obligatoirement) ;
  - Permet d'appeler une méthode de la classe fille qui n'exite pas dans une classe ancètre (mère p.e) ;
  - Exemple : code et trace.

La classe Object

La hiérarchie des classes est décrite par un arbre d'héritage

- La racine de l'arbre est la classe `java.lang.Object` ;
- Chaque classe à la classe `Object` comme super-classe ;
- Cette classe n'a pas d'attribut ;
- Elle possède plusieurs méthodes qui peuvent être redéfinies.

Les méthodes de la classe Object

- `protected Object clone()`, crée et retourne une copie de l'objet ;
- `boolean equals(Object obj)`, teste si `obj` est identique à l'objet ;
- `protected void finalize()`, méthode appelée par le GC lors de la destruction réelle de l'objet ;
- `Class getClass()` retourne la classe d'exécution de l'objet.
- `String toString()`, retourne une representation de l'objet sous la forme d'une chaîne de caractères.

La méthode equal()

- Teste en “surface” si deux objets sont égaux ;
  Un Exemple et sa trace ;
  La méthode `equals()` a testé si `this` avait la même référence que l'objet passé en paramètre ;
- Il faut redéfinir `equals()` pour lui donner un sens ;
  Un Exemple et sa trace.

La méthode toString()

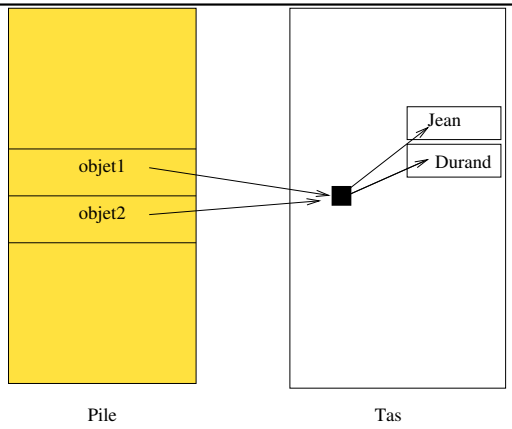
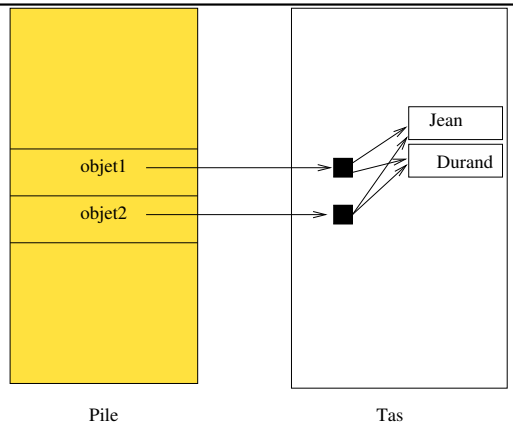
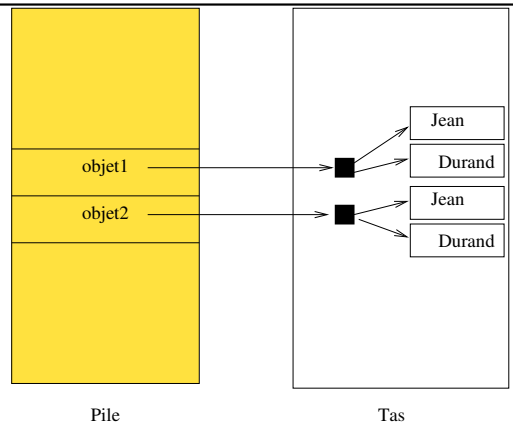
- La méthode `toString()` de la classe `Object` renvoie l'adresse en mémoire de l'instance ;
- On la redéfinit souvent dans les nouvelles classes ;
- `System.out.println(objet)` affiche la chaîne de caractère `objet.toString()` ;
- Utilisée par l'opérateur de concaténation +.
- Un Exemple et sa trace ;

La méthode clone()

- Si on copie une variable qui référence un objet on obtient une autre référence au même objet ;
- On clone un objet lorsqu'on le copie en profondeur ;
- Pourquoi cloner ?
  - Pour protéger un objet ;
  - Pour conserver un objet dans son état initial que l'on réutilisera par la suite.

La méthode clone()

Le problème copie/surface/profondeur

Copie	Clonage de surface	Clonage en profondeur
		
Exemple	Exemple	Exemple
Trace	Trace	Trace

La méthode clone()

Mise en œuvre

Interface Cloneable

- Interface vide ! sert à savoir si un objet peut être cloné ;
- Un objet est donc “clonable” s'il est une instance d'une classe qui implémente l'interface `Cloneable` ;
- Sinon la méthode `clone()` de `Object` renvoie une exception `CloneNotSupportedException`.

La méthode clone()

Mise en œuvre

La classe doit donc :

- Implémenter l'interface `Cloneable` ;
- Redéfinir la méthode `clone()` de façon `public` ;
- Lancer une exception `CloneNotSupportedException` au cas ou le clonage rencontre un objet non clonable.

Partie 6

Entrées/Sorties

Plan de la partie

Entrées/Sorties

2 sortes de communications

1. Entrées/Sorties traditionnelles : clavier, écran et fichiers ;
2. Interaction par l'intermédiaire d'un système de fenêtrage.

Notion de flux

- Un flux est un canal de transmission d'octets non interprétés entre une source et une destination ;
- La source ou la destination peut être :
  - un fichier, une zone mémoire, une URL ...
- Un flux correspond à la notion de canal de transmission sous UNIX ;
- Au sommet de la structure des flux se trouvent quatre classes abstraites : `InputputStream`, `OutputStream`, `Reader` et enfin `Writer` ;
- Les deux premières permettent de manipuler des flux d'octets et les deux autres des flux de caractères.

Utilisation d'un flux

- Le cycle d'utilisation de lecture ou écriture séquentielle d'un flux de données est le suivant :
  1. Ouverture du flux
  2. Tant qu'il y a des données à lire (ou à écrire), on lit (ou on écrit) la donnée suivante dans le flux
  3. Fermeture du flux

Intérêt des flux

- L'intérêt de la notion de flux est qu'elle permet une gestion homogène :
  - quelle que soit la ressource associée au flux de données,
  - quel que soit le flux (y compris `stdin`, `stdout` et `stderr`).

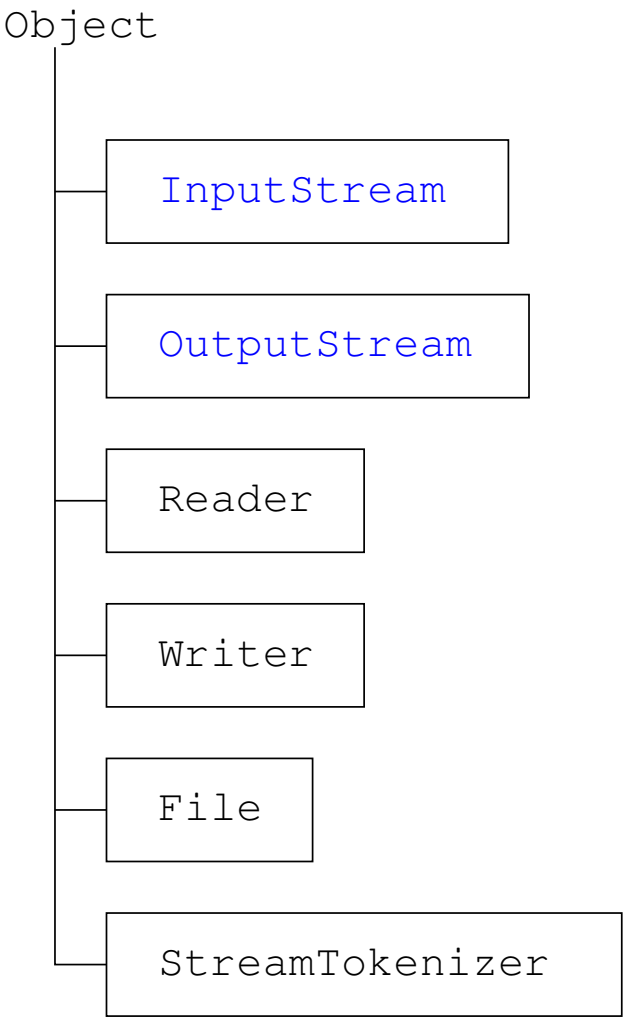
Source ou destination des flux

- Fichier ;
- `Socket` pour échanger des données sur un réseau ;
- Données de grandes tailles dans une base de données (images, par exemple) ;
- Tube entre 2 files d'exécution ;
- Tableau d'octets ;
- Chaîne de caractères ;
- URL (adresse Web)
- ...

Le package `java.io`

- Il contient la plupart des classes liées aux entrée-sorties ;
- Il prend en compte un grand nombre de flux :
  - 2 types de flux (octets et caractères) ;
  - Différentes sources et destinations ;
  - Utilisation de filtres de lecture ou d'écriture.
- On combine donc flux et filtres pour réaliser la gestion souhaitée pour les E/S.

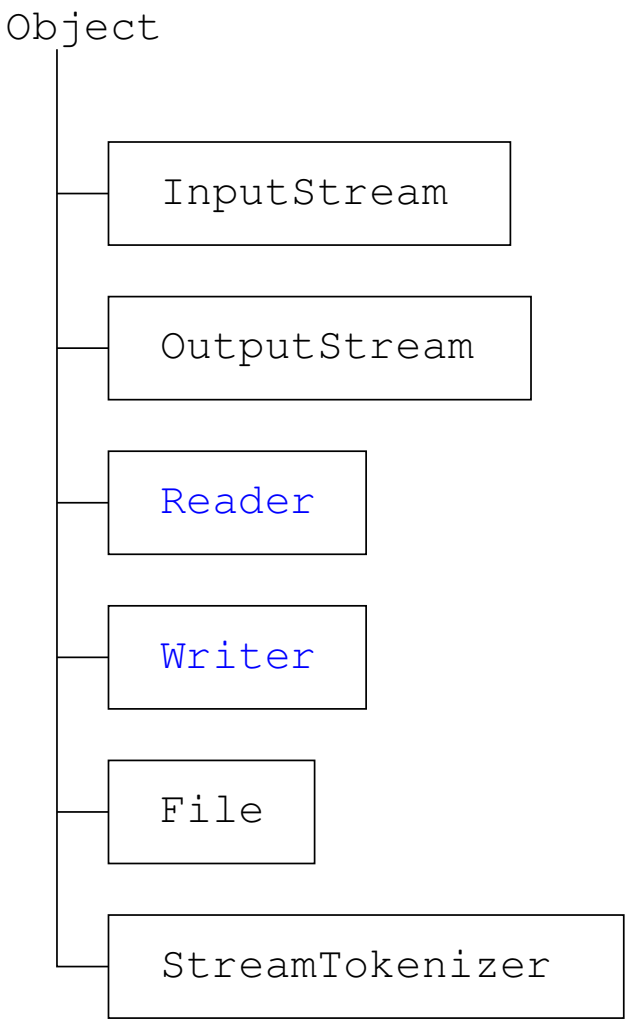
Hierarchie des classes



Les flux d'octets

- Classes abstraites `InputStream` et `OutputStream` et leurs sous-classes concrètes respectives ;
- Servent à lire ou écrire des octets « bruts » qui représentent des données codées, manipulées par un programme.

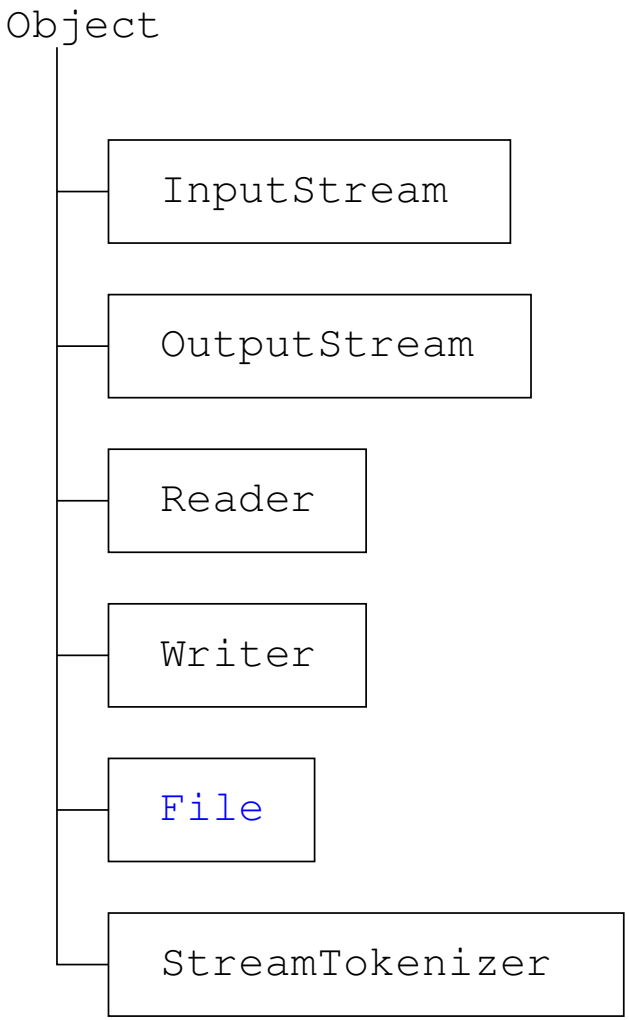
Hierarchie des classes



Les flux de caractères

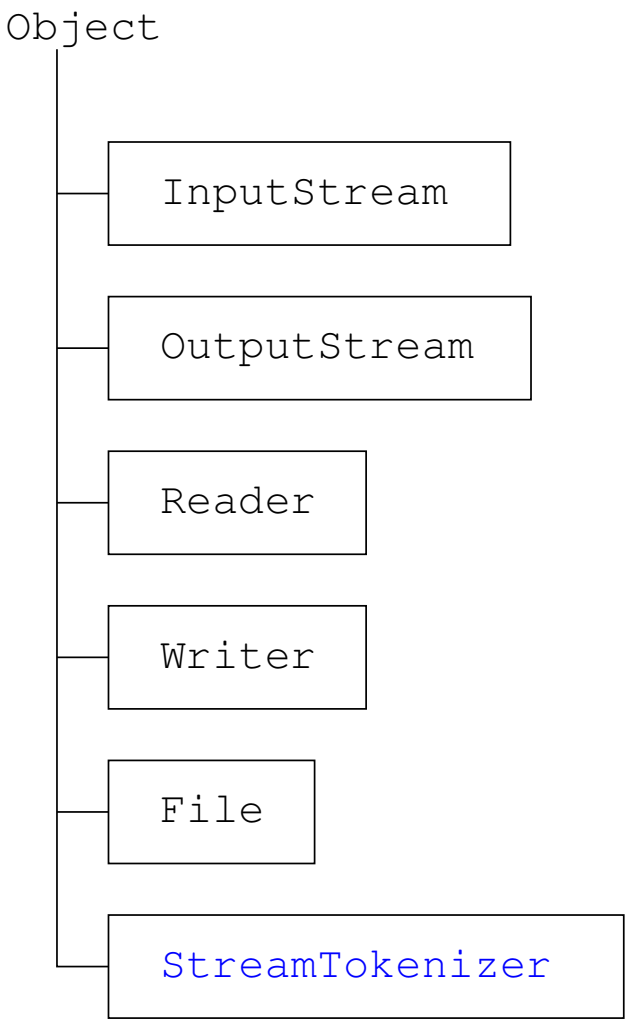
- Classes abstraites `Reader` et `Writer` et leurs sous-classes concrètes respectives ;
- Servent à lire ou écrire des données qui représentent des caractères lisibles, codés en Unicode.

Hierarchie des classes



Maniement des fichiers et des répertoires

Hierarchie des classes



Analyse lexicale d'un flux d'entrée

Utilisation des filtres

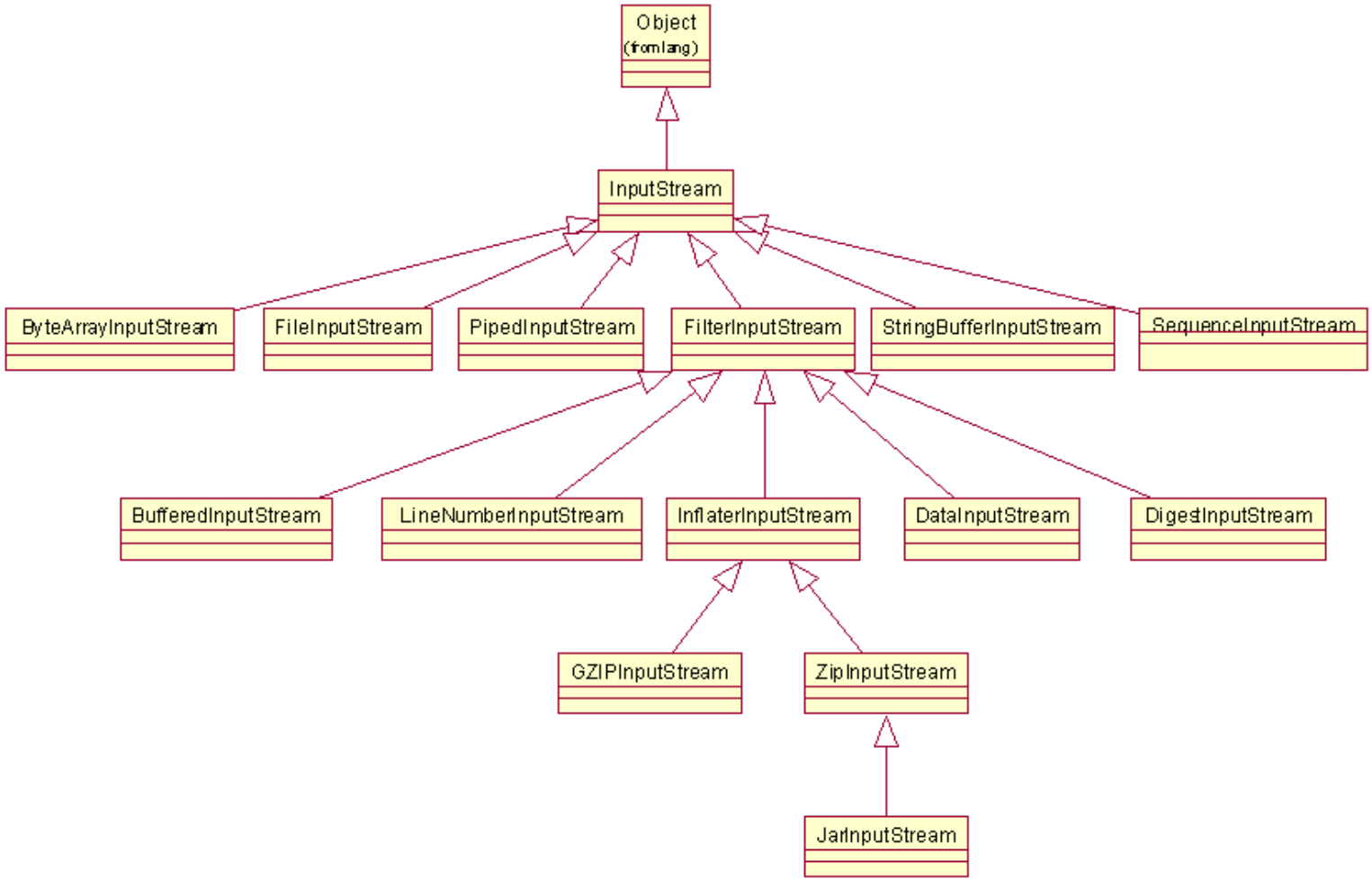
- Les fonctionnalités de base d'un flux sont la lecture ou l'écriture (méthodes `read()` ou `write()`)
- Selon les besoins, on peut lui ajouter des filtres :
  - Utilisation d'un buffer pour réduire les lectures ou écritures « réelles » ;
  - Codage ou décodage des données manipulées ;
  - Compression ou décompression de ces données ;
  - etc. . ....

Utilisation des filtres

Exemple, on utilise `BufferedReader` qui ajoute un buffer pour lire un flux de caractères

```
FileReader r = new FileReader(fichier);
BufferedReader br = new BufferedReader(r);
                  // br utilise un buffer pour lire
int c;
try {
     while ((c = br.read()) != -1) . . .
    }
```

Flux d'octets en entrée



La classe java.io.InputStream

- Classe abstraite ;
- C'est la racine des classes qui concernent la lecture d'octets depuis un flux de données ;
- « Interface » selon laquelle sont vues toutes les classes de flux qui lisent des octets ;
- Elle possède un constructeur sans paramètre ;
- Les méthodes :

```
// Constructeur
public InputStream()
// Les méthodes publiques
abstract int read() throws IOException
public int read(byte[] buf) throws IOException
public int read(byte[] buf, int debut, int nb) throws IOException
public long skip(long n) throws IOException
public int available() throws IOException
public void close() throws IOException
synchronized void mark(int nbOctetsLimite)
synchronized void reset() throws IOException
public boolean markSupported()
```

Description des méthodes

- `int read()`
  - Renvoie l'octet lu dans le flux (sous forme d'un entier compris entre 0 et 255), ou -1 si la fin du flux est rencontré ;
  - Bloque jusqu'à la lecture d'un octet, ou la rencontre de la fin du flux, ou d'une exception (comme toutes les autres méthodes `read`)
  - Méthode abstraite
- `int read(byte[] buf)`
  - Essaie de lire assez d'octets pour remplir le tableau `buf` ;
  - Renvoie le nombre d'octets réellement lus (elle est débloquée par la disponibilité d'au moins un octet), ou -1 si elle a rencontré la fin du flux ;
  - Implémentée en utilisant la méthode `read()` (à redéfinir dans les classes filles pour de meilleures performances).

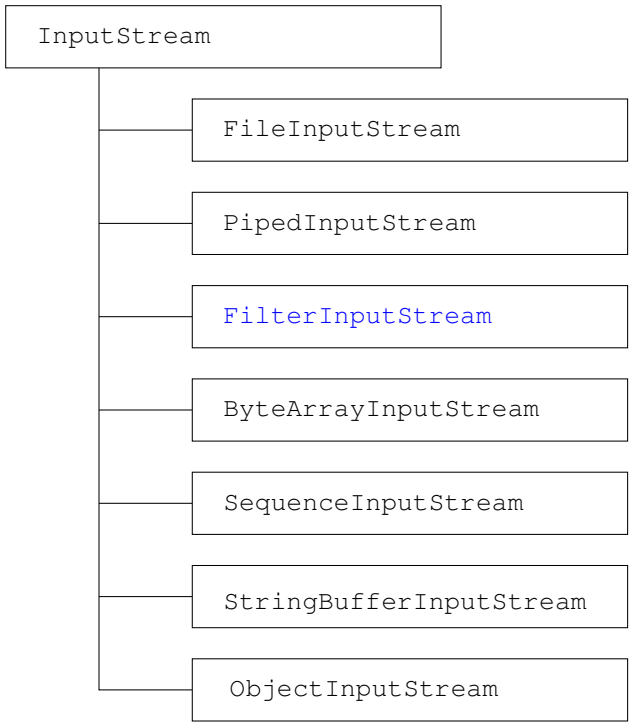
Description des méthodes

- `int read(byte[] buf, int debut, int nb)`
  - Lit `nb` octets et les place dans le tableau `buf` à partir de l'indice `debut` (de `debut` à `debut + nb - 1`) renvoie le nombre d'octets lus, ou -1 si elle a rencontré la fin du flux ;
- `long skip(long n)`
  - Saute `n` octets dans le flux ;
  - Renvoie le nombre d'octets réellement sautés ;
- `int available()`
  - Renvoie le nombre d'octets prêts à être lus.

Description des méthodes

- `boolean markSupported()`
  - Indique si le flux supporte la notion de marque pour revenir en arrière dans la lecture ;
- `void mark(int nbOctetsLimite)`
  - Marque la position actuelle pour un retour ultérieur éventuel à cette position avec `reset` ;
  - `nbOctetsLimite` indique le nombre d'octets lus après lequel la marque peut être « oubliée » ;
- `void reset()`
  - Positionne le flux à la dernière marque.

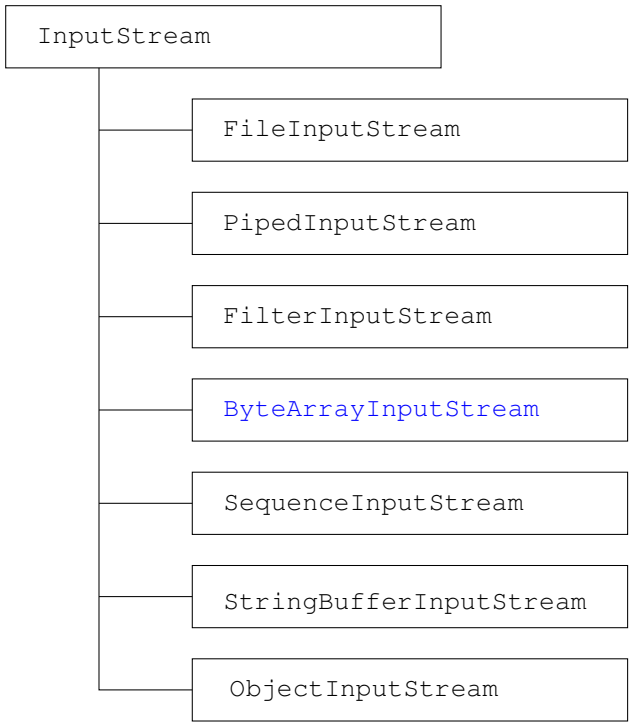
Quelques classes dérivées



Mère des classes des filtres.

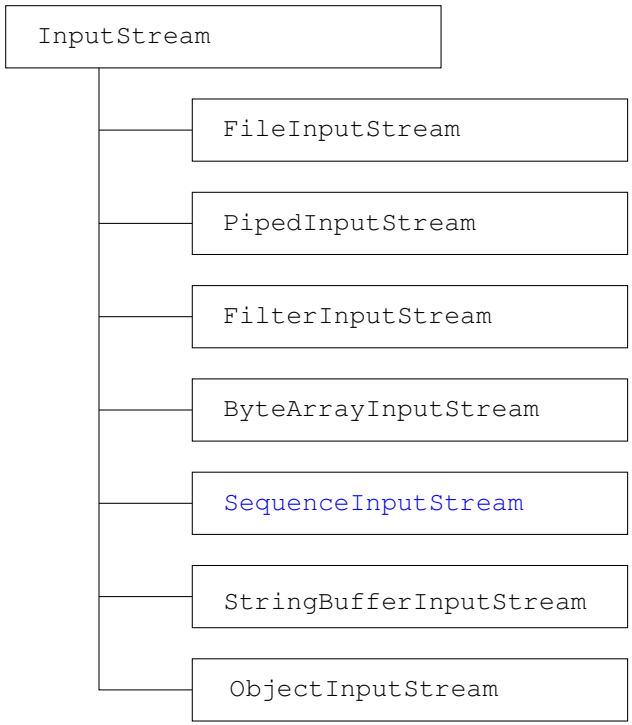
- `DataInputStream`
- `BufferedInputStream`
- `PushbackInputStream`
- `LineNumberInputStream`

Quelques classes dérivées



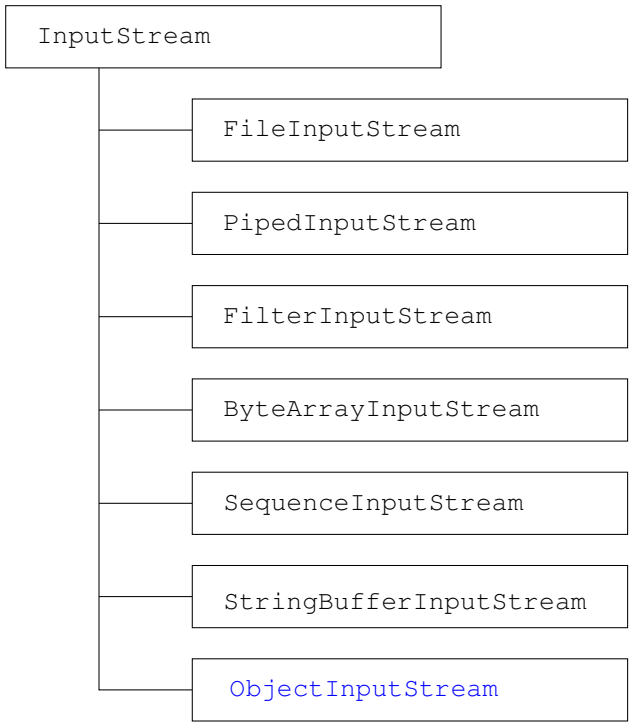
Lecture depuis un tableau.

Quelques classes dérivées



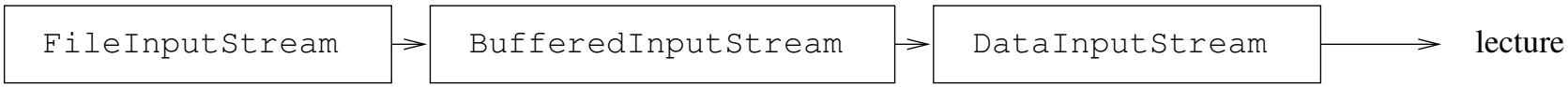
Concaténation des flux d'entrée.

Quelques classes dérivées



Désérialisation des flux.

Exemple commenté



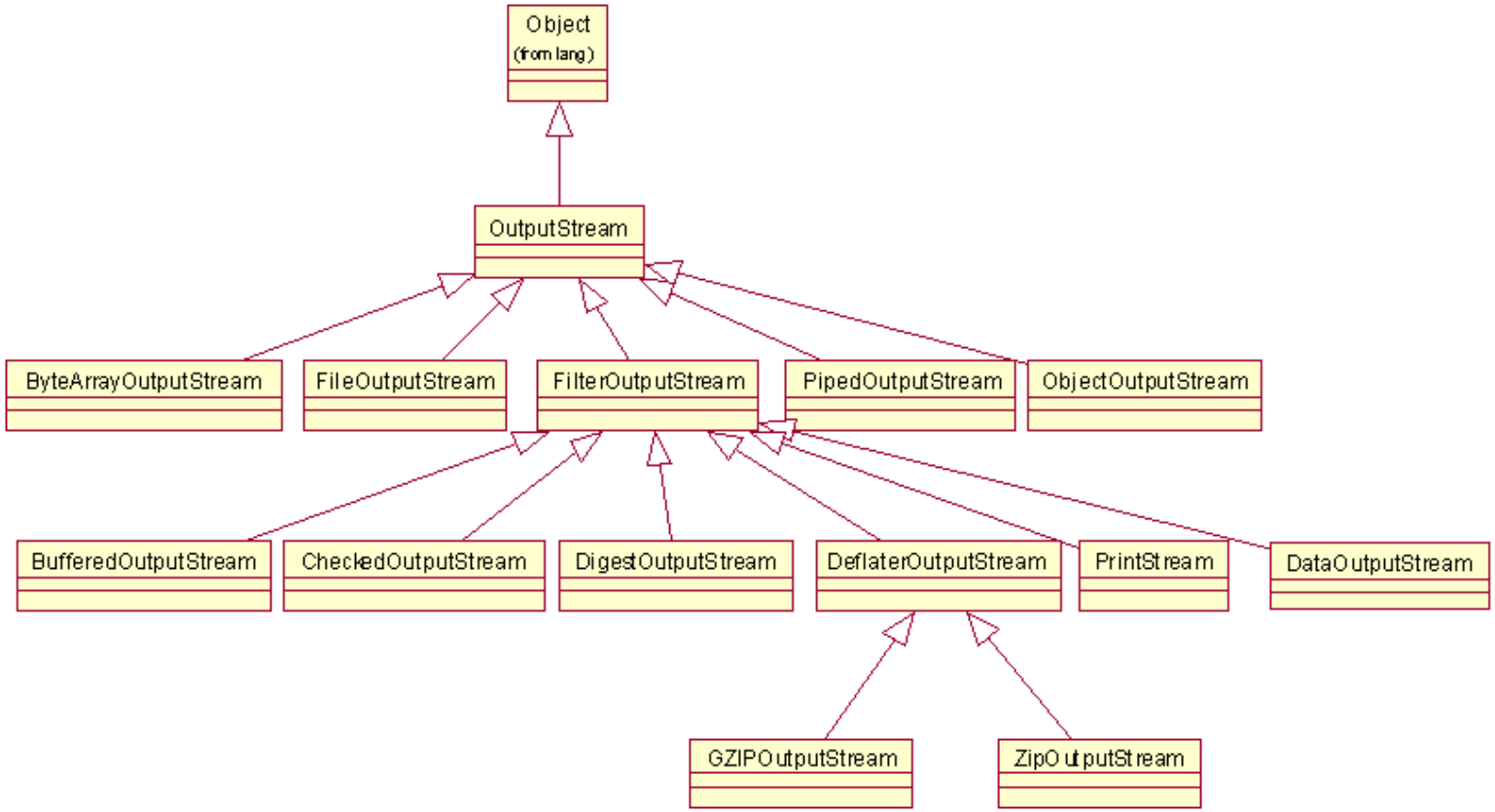
- `FileInputStream`
  - Classe de base pour la lecture d'un fichier ;
- filtré par un `BufferedInputStream`
  - Classe fille de la classe `FilterInputStream` ;
  - Ajoute un buffer pour la lecture du flux ;
- filtré par un `DataInputStream`
  - Classe fille de la classe `FilterInputStream` ;
  - Filtre qui décode les types primitifs Java codés dans un format standard, indépendant du système.

Exemple commenté

```
FileInputStream fis =
  new FileInputStream("fichier");
BufferedInputStream bis =
  new BufferedInputStream(fis);
DataInputStream dis =
  new DataInputStream(bis);

double d = dis.readDouble();
String s = dis.readUTF();
int i = dis.readInt();
dis.close();
//...
```

Flux d'octets en sortie

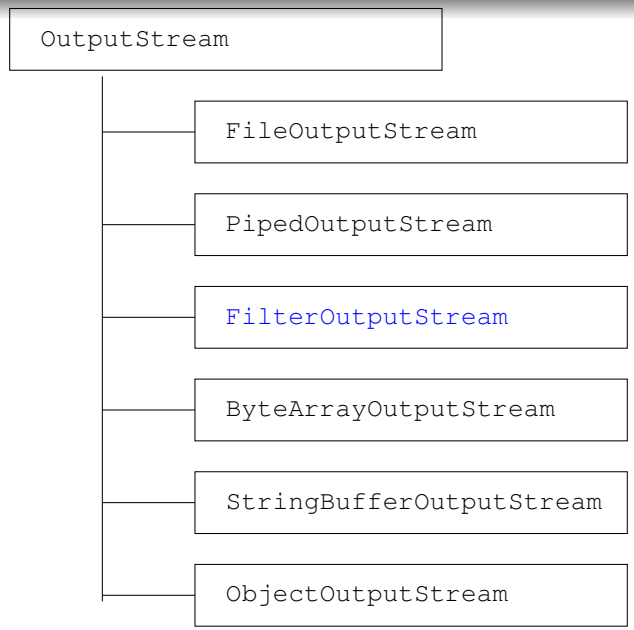


La classe java.io.OutputStream

- Classe abstraite ;
- Fournit des méthodes de manipulation de flux en sortie ;
- Ces méthodes permettent d'écrire des données sous forme de tableaux de bytes ;
- Les méthodes :

```
// Constructeur
public OutputStream()
// Les méthodes publiques
abstract void write(int byte) throws IOException
  // Ecrit un byte sur le flux
public abstract int write(byte[] buf) throws IOException
  // Ecrit un tableau de bytes sur le flux
public abstract int write(byte[] buf, int off, int len) throws IOException
  //Ecrit un tableau de bytes à partir de off de longueur len sur flux.
public void close() throws IOException
```

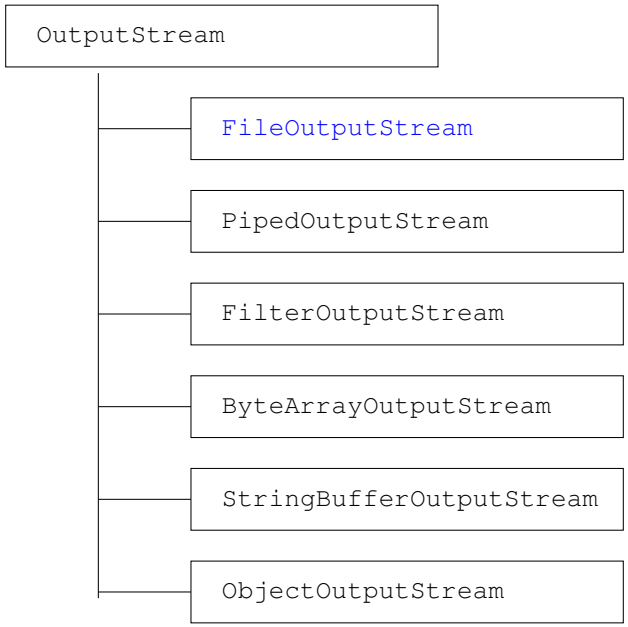
Quelques classes dérivées



Mère des classes des filtres.

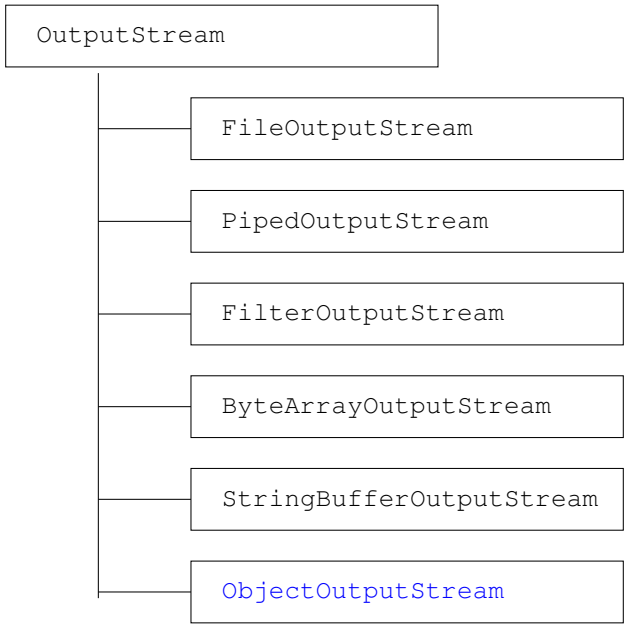
- `DataOutputStream` Écriture de type primitif ;
- `BufferedOutputStream` Sortie bufferisée ;
- `PrintStream` utilisé par `system.out` ne pas utiliser autrement.

Quelques classes dérivées



Écriture d'octets

Quelques classes dérivées



Écriture d'objets sérialisés

Écriture dans un fichier

```
import java.util.*;
import java.io.*;

public class TestFileOutputStream {

    public static void main(String args[]){
  TestFileOutputStream tfos = new TestFileOutputStream();
  tfos.cree_file(args[0]);
    }

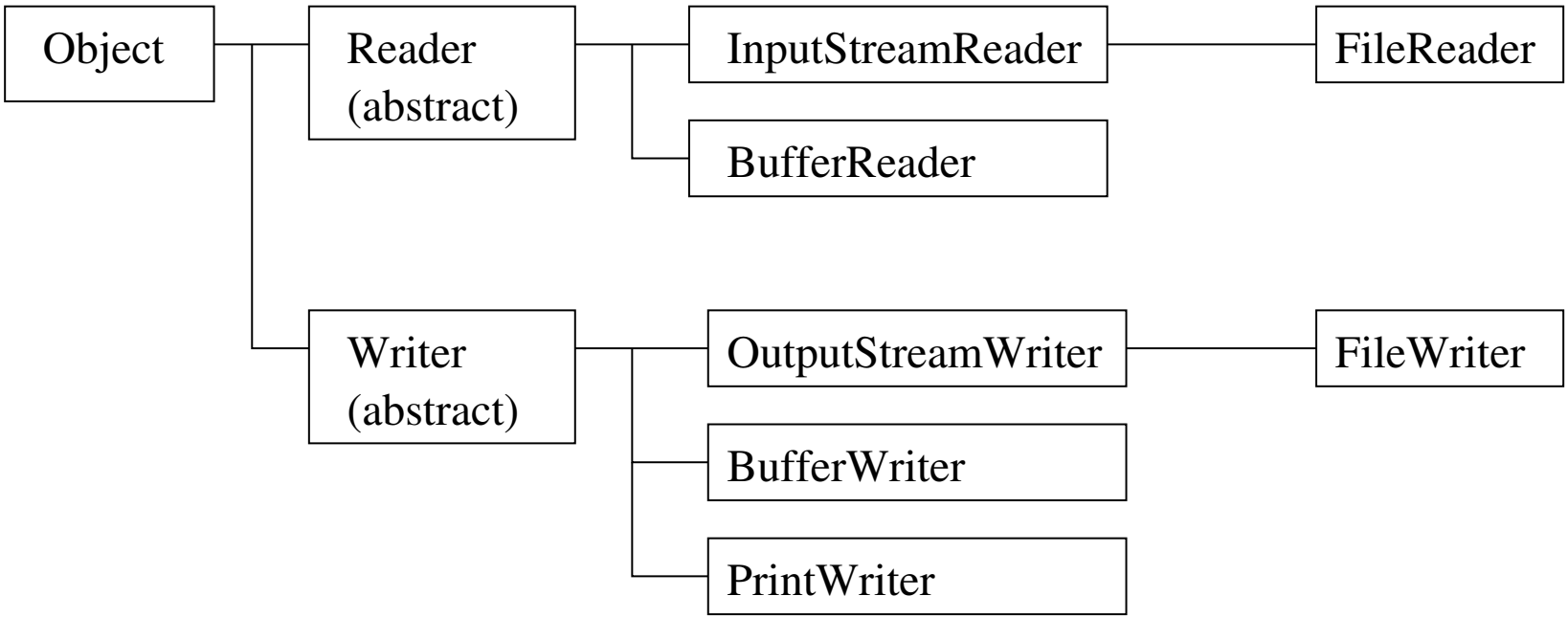

    void cree_file(String nomf){
        File fic = new File(nomf);
  System.out.print("Entrez votre texte : ");
        Scanner sc = new Scanner(System.in);
  String s = sc.next();
  try{
      FileOutputStream fos= new FileOutputStream(fic);
      int l;
      byte b[] = new byte[l=s.length()];
      for (int i=0; i<l; i++) b[i]= (byte) s.charAt(i);
      fos.write(b);
      fos.close();
  } catch(IOException e2){System.out.println("Err I/O "+e2);}
    }
}// Fin de la class TestFileOutputStream
```

Une version plus sophistiquée.

La classe `java.io.PrintStream`

- Cette classe possède les 2 méthodes `print()` et `println()` qui écrivent tous les types de données sous forme de chaînes de caractères
- Aucune des méthodes de `PrintStream` ne lève d'exception ;
- On peut savoir s'il y a eu une erreur en appelant la méthode `checkError()` ;
- Attention, `println()` n'effectue un `flush()` (vidage des buffers) que si le `PrintStream` a été créé avec le paramètre `autoflush`.

Classes de gestion de flux de caractères



- `Reader` et `Writer` : flux de caractères (texte Unicode) ;
- `BufferedReader` et `BufferedWriter` : flux de buffer important.

Saisies clavier

- `System.in` désigne l'entrée standard : le clavier ;
- `int System.in.read() throws IOException`
  - lecture octet par octet ;
  - renvoie code ASCII du caractère lu ;
  - renvoie (-1) si en fin de flux, cas d'une lecture fichier.
- `String readLine()` méthode de la classe `BufferedReader` : permet la lecture d'une chaîne de caractères jusqu'au saut de ligne.

Saisies clavier - exemple

```
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 kr. : ");
      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.parseFloat(line)
     ;
      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");}
  }
}
```

Lecture d'un fichier

- `FileReader` va désigner un flux d'entrée. Son constructeur (qui peut déclencher l'exception `FileNotFoundException`) aura pour paramètre une chaîne de caractères désignant le nom du fichier à partir duquel se fera l'entrée.
- La méthode `close()` permet de fermer le fichier (peut déclencher l'exception `IOException`).

Lecture d'un fichier

```
import java.io.*;
public class LireLigne {
  public static void main(String[] args)
    {
      try
       {
         FileReader fr=new FileReader("/etc/fstab");
         BufferedReader br= new BufferedReader(fr);
         while (br.ready())
            System.out.println(br.readLine());
         br.close();
        }
     catch (Exception e)
        {System.out.println("Erreur "+e);}
    }
}
```

Lecture d'un fichier - exemple

```
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.parseFloat(line);
    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) {}
    }
  }
}
```

Ecriture dans un fichier

- On utilise la classe `PrintWriter` qui implémente des écritures formattées de type texte et analogue à l'affichage écran.
- Exemple :

```
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 le fichier out.txt");
   out.close();
  }
}
```

Analyse lexicale

- On peut lire et écrire des données brutes (non formattées comme du texte) avec les classes `DataInputStream` et `DataInputStream` : fichier non lisible, mais de taille réduite.
- `StringTokenizer` est une classe permettant d'instancier un petit analyseur qui peut, par exemple, découper des lignes de caractères et en récupérer les nombres qu'elle contient, en identifiant les séparateurs.
- La classe `java.io.File` permet de faire des manipulations de fichiers (lister 1 répertoire, renommer ou supprimer un fichier, ...)

Analyse lexicale

Lecture avec analyseur

Utilisation de `StringTokenizer`

- On lit chaque ligne avec `readLine()` ;
- On construit un objet `StringTokenizer` qui récupère la chaîne de caractères lue ;
- On récupère les tokens séparés par des blancs (`nextToken()`).

```
import java.io.*;
import java.util.*;
public class LectureParse {
public static void main(String[] args)
{
 try {
      FileReader fic =
         new FileReader("ficdonnees.
     dat");
      BufferedReader lecBuf =
         new BufferedReader(fic);
      StringTokenizer st;
      String line, word;
      double x, y, z;
      while((line = lecBuf.readLine())
      != null)
      {
       st = new StringTokenizer(line);
       word = st.nextToken();
       x = Double.parseDouble(word);
       word = st.nextToken();
       y = Double.parseDouble(word);
       word = st.nextToken();
       z = Double.parseDouble(word);
       System.out.println
          (``Coordonnees :'' + x +' '+
      y + ' ' +z);
      }
     }
catch(Exception e){}
```

Lecture du contenu d'un répertoire

```
import java.io.*;
class ListeRepertoire {
  public static void main(String args[]) throws Exception
  {
   File repertoire = new File(args[0]);
   if (!repertoire.exists() || !repertoire.canRead())
     {
      System.out.println(repertoire + ``non lisible'');
      return;
     }
   if (repertoire.isDirectory())
     {
      String [] fichiers = repertoire.list();
      for(int i=0; i<fichiers.length; i++)
        System.out.println(fichiers[i]);
     }
  }
}
```

Les exceptions

Principales exceptions :

- `Exception`
  - `IOException`
    exception durant une entrée sortie
    - `EOFException`
      lecture d'une fin de fichier
    - `FileNotFoundException`
      fichier non trouvé
    - `ObjectStreamException`
      problème lié à la sérialisation

Autres type de flux

- La classe `java.io.RandomAccesFile`
  - Elle permet l'ouverture de fichiers en lecture et écriture ;
  - Elle permet l'accès direct aux fichiers (`seek`) ;
  - Elle offre les fonctionnalités de `java.io.InputStream` (`read`) et `java.io.OutputStream` (`write`) par l'implémentation de ces interfaces ;
  - Elle offre les fonctionnalités de `DataInputStream` (`readXXX`) et de `DataOutputStream` (`writeXXX`) réunis, sans en hériter.

Accès direct aux fichiers

Les constructeurs et les méthodes de `java.io.RandomAccesFile`

```
// Les constructeurs
public RandomAccessFile(String name, String mode) throws IOException
public RandomAccessFile(File file, String mode) throws IOException
   // Mode vaut "r" pour Lecture ou "rw" pour Lect/Ecriture
// Les méthodes publiques
int read() throws IOException
  // Lit un octet et le retourne
public int read(byte[] buf) throws IOException
  // Lit dans un tableau d'octets
public int read(byte[] buf, int off, int len) throws IOException
  //Lit dans buf à partir de off, len bytes.
public final boolean readBoolean() throws IOException
public final byte readByte() throws IOException
public final char readChar() throws IOException
public final double readDouble() throws IOException
public final float readFloat() throws IOException
public final void readFully(byte []) throws IOException
public final void readFully(byte [], int off, int len) throws IOException
public final int readInt() throws IOException
public final String readLine() throws IOException
public final long readLong() throws IOException
public final short readShort() throws IOException
public final int readUnsignedByte() throws IOException
public final int readUnsignedShort() throws IOException
public final int skipByte(int n) throws IOException
```

Accès direct aux fichiers

Les constructeurs et les méthodes de `java.io.RandomAccesFile`

```
// Les méthodes public
abstract void write(int byte) throws IOException
  // Ecrit un byte sur le flux
public abstract int write(byte[] buf) throws IOException
  // Ecrit un tableau de bytes sur le flux
public abstract int write(byte[] buf, int off, int len) throws IOException
  //Ecrit un tableau de bytes à partir de off de long len
public final void writeBoolean(boolean v) throws IOException
public final void writeByte(int v) throws IOException
public final void writeBytes(String s) throws IOException
public final void writeChar(int v) throws IOException
public final void writeChars(String s) throws IOException
public final void writeDouble(double v) throws IOException
public final void writeFloat(float v) throws IOException
public final void writeInt(int v) throws IOException
public final void writeLong(long v) throws IOException
public final int writeShort(int v) throws IOException
public int skipBytes(int n) throws IOException
  //permet de sauter n bytes
public void seek(long pos) throws IOException
  //Positionne le pointeur de fichier à pos public
void close() throws IOException
```

Partie 7

Interfaces graphiques

Plan de la partie

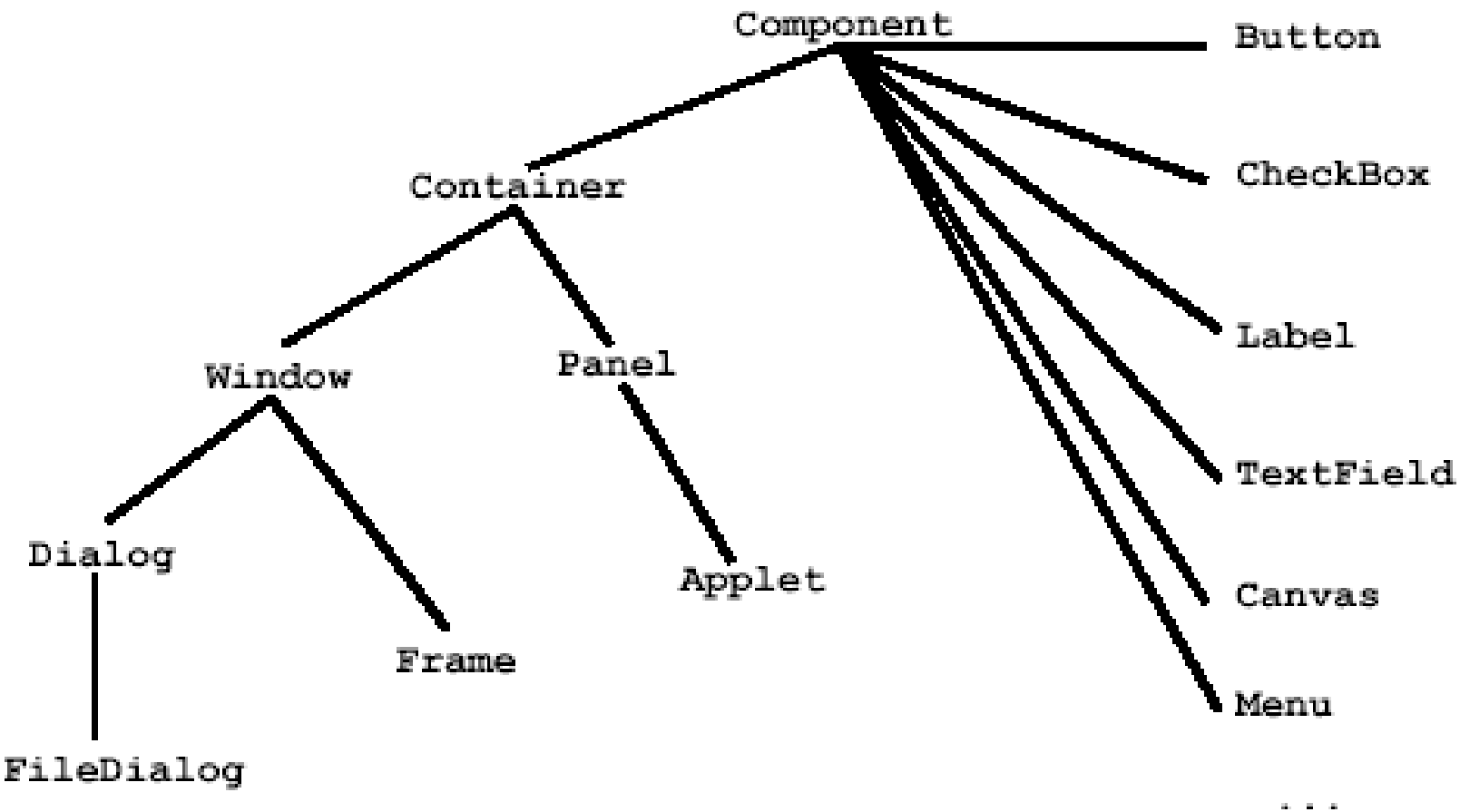
Abstract Window Toolkit (AWT)

- Pour atteindre le but de portabilité fixé dans Java, l'AWT fournit la couche d'interfaçage graphique portable :
  - Éléments graphiques usuels : boutons, menus,
  - utilisables dans les applets ou les applications
  - S'adaptent au visuel des différents environnements car bâtis avec les fonctions native Windows, Unix, MacOS ...
  - Dérivables pour les adapter à ses propres besoins
  - Des protocoles d'agencement de composant graphiques sont fournis.
- Contient la définition conceptuelle des objets :
  - Leur comportement et leur spécification
  - Pas leur aspect graphique qui dépend de la plate-forme d'exécution
  - L'inconvénient majeur en tentant d'être portable, l'AWT se prive de certaines fonctions graphiques avancées mais la librairie de classes graphiques SWING pallie cet inconvénient.

Les composants de base de l'AWT

- Les principaux types de composants :
  - Les conteneurs (container) : peuvent contenir des canevas, des composants graphiques, **ou d'autres conteneurs** ;
  - Les canevas (canvas) : objets graphiques très simples, sur lesquels on ne peut que dessiner et afficher des images (on dit aussi "toile de fond" ou "fond")
  - Les composants graphiques "sensibles" : objets traditionnels présents dans les interfaces graphiques leur comportement est déjà connu et établi

Hierarchie partielle des classes



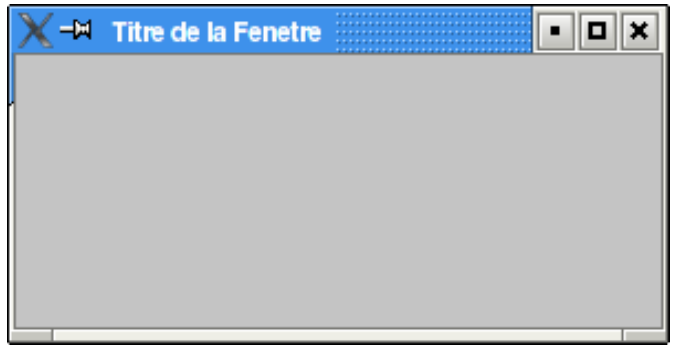
Ne pas confondre avec la hiérarchie interne à chaque application qui définit l'imbrication des différents composants graphiques.

Les conteneurs

- `Container` : Gestionnaire de composants ;
- `Window` : Une fenêtre sans bordure ;
- `Frame` : Une fenêtre avec bordure ;
- `Dialog` : Une boite de dialogue
- `FileDialog` : Une boite de dialogue pour sélectionner un fichier ;
- `Panel` : un conteneur "fourre-tout".

Exemple de frame

```
import java.awt.*;
public class MaFrame extends Frame {
  public MaFrame() {
    super();
    setTitle(" Titre de la Fenetre ");
    setSize(300, 150);
    setVisible(true); // affiche la fenetre
  }
  public static void main(String[] args) {
    new MaFrame(); }
}
```

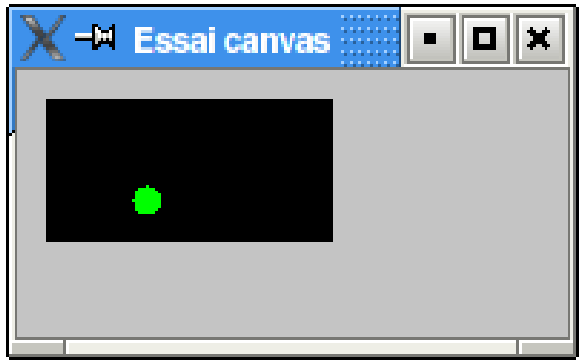


Exemple de Canvas

```
import java.awt.*;

class UnCanvas extends Canvas
{
 public void paint(Graphics g) {
  g.setColor(Color.black);
  g.fillRect(10, 10, 100,50);
  g.setColor(Color.green);
  g.fillOval(40, 40, 10,10);
 }
}

public class TestCanvas  {
 static public void main (String arg [
      ]) {
  Frame w = new Frame("Essai canvas");
  Canvas c = new UnCanvas();
  c.setSize(new Dimension(200,200));
  w.add("Center",c);
  w.setSize(200,120);
  w.setVisible(true);
  c.paint(c.getGraphics());
 }
}
```



Utilisation des conteneurs

- Le rôle pricipal est de contenir des composants graphique qui peuvent eux-mêmes contenir un *conteneur* ;
- Un composant graphique doit toujours être incorporé dans un conteneur.

Un conteneur possède deux méthodes fondamentales :

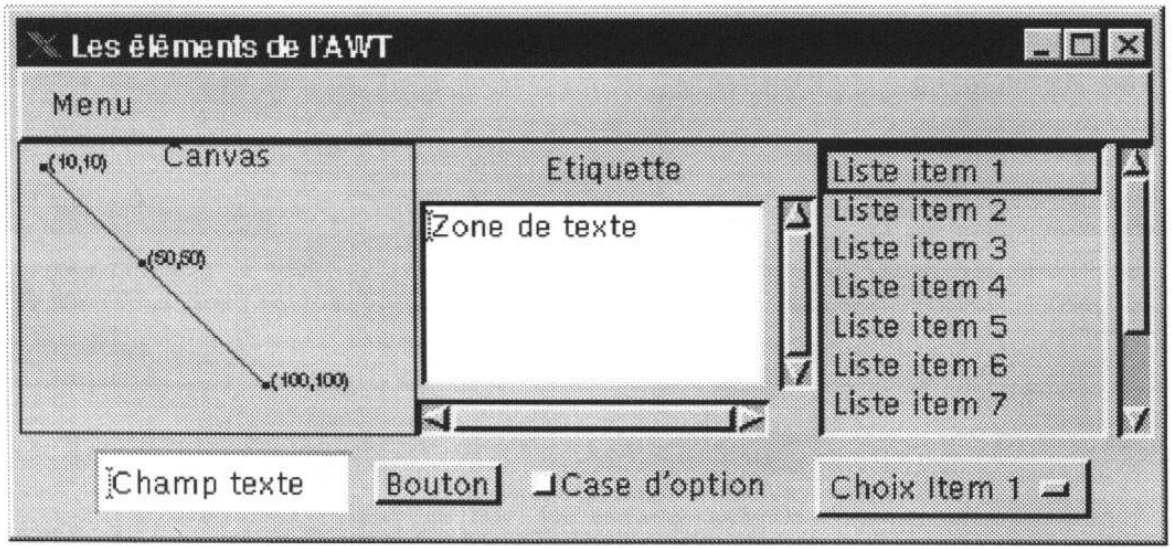
- add(*leComposant)* cette méthode peut nécessiter d'autres arguments
  - Un composant est donc ajouté dans le conteneur.
- Associer au conteneur un gestionnaire de disposition ;

Les composants de l'AWT

Graphics User Interface

Plusieurs éléments peuvent apparaître :

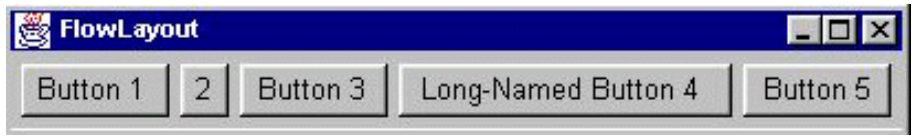
- `Canvas`, pour tracer des figures géométriques ;
- Etiquette `Label` pour afficher un texte ;
- Zone de texte modifiable `TextArea` ;
- Liste déroulante à choix `List` ;
- Zone de saisie de texte `TextField` ;
- Bouton `Button` ;
- Case d'option `Checkbox` ;
- Menu déroulant `Menu et MenuBar`.



Gestionnaire de placement

Les différents composants d'une fenêtre peuvent y être placés de plusieurs manières avec un gestionnaire de placements (layout manager) :

- `FlowLayout` range les composants ligne par ligne et de gauche à droite ;
- `BorderLayout` place les composants dans 5 zones : le centre et les 4 côtés ;
- `GridLayout` place les composants sur une grille 2D ;
- `CardLayout`
- `GridBagLayout` place les composants sur une grille 2D, avec des coordonnées. Les zones ne sont toutes de même dimension.



Utilisation du gestionnaire de placement

- Chaque Container possède un gestionnaire de placement (Layout Manager) par défaut ;
- Ce gestionnaire d'agencement définit la méthode de placement des composants d'un Container les uns par rapport aux autres ;
- Chaque conteneur est associé à un et un seul layout manager
  - Tous ses composants sont soumis à ce protocole de mise en forme ;
  - Par contre, parmi ces composants peuvent se trouver d'autres conteneurs qui eux sont soumis à d'autres gestionnaires.

Différents gestionnaires de placement

Java permet de définir des modèles de mise en page des composants avec la classe `java.awt.LayoutManager`. L'association du gestionnaire de mise en page avec le container se fait par la méthode `setLayout` de l'objet container. Les différents modèles de mise en page sont :

- `FlowLayout` : modèle par défaut qui positionnne les composants sur une même ligne ou sur la ligne suivante si la ligne courante est remplie.
- `BorderLayout` : modèle de placement des composants en 4 bords (`North, South, East, West`) et un centre (`Center`). Le placement des composants se fera en ajoutant une de ces situations en premier paramètre de la méthode `add()`. Par exemple : `bouton.add("West","mon bouton")`.
- `GridLayout(int nl, int nc)` : est un modèle de placement de composants dans un container découpé régulièrement en `nl` lignes et `nc` colonnes.

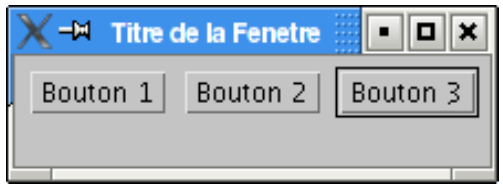
FlowLayout

- Les composants graphiques sont ajoutés :
  - Les uns après les autres ;
  - De gauche à droite
  - Avec un saut de ligne dès qu'il ne reste plus de d'espace suffisant à droite.
- Les constantes d'alignement :
  - `FlowLayout.CENTER` (défaut) ;
  - `FlowLayout.LEFT` ;
  - `FlowLayout.RIGHT`.

FlowLayout, un exemple

```
import java.awt.*;

public class LayoutFlow extends Frame
      {
  public LayoutFlow() {
   super(); setTitle(" Titre de la
     Fenetre ");
   setSize(300, 150); setLayout(new
     FlowLayout());
   add(new Button("Bouton 1"));
   add(new Button("Bouton 2"));
   add(new Button("Bouton 3"));
   pack(); setVisible(true); //
     affiche la fenetre
  }
  public static void main(String[]
     args) {
  new LayoutFlow();
  }
}
```

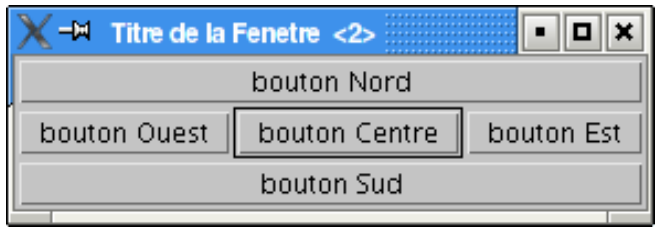


BorderLayout

- 5 zones (`North`, `South`, `East`, `West` et `Center`) qui pourront recevoir au plus un composant ;
- Attention aux problèmes de superposition de composants.

BorderLayout, un exemple

```
import java.awt.*;
public class LayoutBorder extends
      Frame {
 public LayoutBorder() {
  super();
  setTitle(" Titre de la Fenetre ");
  setSize(300, 150);
  setLayout(new BorderLayout());
  add("North", new Button(" bouton
      Nord "));
  add("South", new Button(" bouton
      Sud "));
  add("West", new Button(" bouton
      Ouest "));
  add("East", new Button(" bouton Est
       "));
  add("Center", new Button(" bouton
      Centre "));
  pack(); setVisible(true);
 }
 public static void main(String[]
      args) {
  new LayoutBorder();
 }
}
```



GridLayout

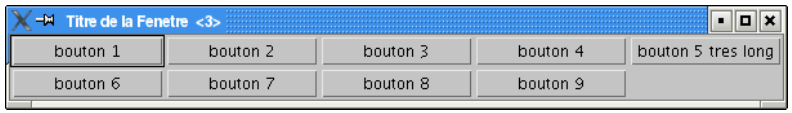
- Spécifie une grille dont chaque cellule peut contenir un composant graphique ;
  - Les composants sont ajoutés : de gauche à droite, ligne par ligne ;
  - Si on spécifie pas le nombre de colonne et de ligne, le placement se fait sur une seule ligne ;
  - Le nombre de ligne ou de colonne peut être égal à 0 ;
  - Les cellules de la grille sont toutes de la même taille.

GridLayout, un exemple

```
import java.awt.*;

public class LayoutGrid extends Frame
      {
 public LayoutGrid() {
  super();
  setTitle(" Titre de la Fenetre ");
  setSize(300, 150);
  setLayout(new GridLayout(2, 0));
  add(new Button("bouton 1"));
  add(new Button("bouton 2"));
  add(new Button("bouton 3"));
  add(new Button("bouton 4"));
  add(new Button("bouton 5 tres long"
     ));
  add(new Button("bouton 6"));
  add(new Button("bouton 7"));
  add(new Button("bouton 8"));
  add(new Button("bouton 9"));
  pack(); setVisible(true);
 }

 public static void main(String[]
     args) {
  new LayoutGrid();
 }
}
```



CardLayout

- Aide à construire des boîtes de dialogue composées de plusieurs onglets ;
- Un onglet se compose généralement de plusieurs contrôles :
  - On insère des panneaux dans la fenêtre utilisée par le CardLayout Manager.
  - Chaque panneau correspond à un onglet de boîte de dialogue et contient plusieurs contrôles.
  - Par défaut, c'est le premier onglet qui est affiché.

CardLayout, un exemple

```
import java.awt.*;

public class LayoutCard extends Frame
      {
 public LayoutCard() {
  super();
  setTitle("Titre de la Fenetre ");
  setSize(300,150);
  CardLayout cl = new CardLayout();
  setLayout(cl);
  //création d'un panneau contenant
     les contrôles d'un onglet
  Panel p = new Panel();
  //ajouter les composants au panel
  p.add(new Button("Bouton 1 panneau
     1"));
  p.add(new Button("Bouton 2 panneau
     1"));
  // inclure le panneau dans la fenê
     tre
  // sous le nom "Page1"
  add("Page1",p);
  //déclaration et insertion de l'
     onglet suivant
  p = new Panel(); p.add(new Button("
     Bouton 1 panneau 2"));
  add("Page2", p);
  // affiche la fenetre
  pack(); setVisible(true);
 }

 public static void main(String[]
```

GridBagLayout

- Le conteneur est divisé en cellules égales.
- Un composant peut occuper plusieurs cellules de la grille.
- Il est possible de faire une distribution dans des cellules distinctes.
- Un objet de la classe `GridBagConstraints` permet de donner les indications de positionnement et de dimension à l'objet `GridBagLayout`.
  - Les lignes et les colonnes prennent naissance au moment ou les contrôles sont ajoutés.
  - Chaque contrôle est associé à un objet de la classe `GridBagConstraints` qui indique l'emplacement voulu pour le contrôle.

GridBagLayout, un exemple

```
import java.awt.*;

public class LayoutGridBag extends
      Frame {
  public LayoutGridBag() {
   super();
   setTitle(" Titre de la Fenetre ");
   setSize(300, 150);
   Button b1 = new Button(" bouton 1
     ");
   Button b2 = new Button(" bouton 2
     ");
   Button b3 = new Button(" bouton 3
     ");
   GridBagLayout gb = new
     GridBagLayout();
   GridBagConstraints gbc =
     new GridBagConstraints();
   setLayout(gb);
   gbc.fill = GridBagConstraints.BOTH
     ;
   gbc.weightx = 1;
   gbc.weighty = 1;
   gbc.gridx = 1;
   gb.setConstraints(b1, gbc);
   gb.setConstraints(b2, gbc);
   gb.setConstraints(b3, gbc);
   add(b1); add(b2); add(b3);
   pack(); setVisible(true);
  }
  public static void main(String[]
     args) {
```

Evénement

- Les interactions (provenant de la souris ou du clavier) sont gérées comme des événements fournis à un objet graphique qui peut être :
  - Un composant graphique "sensible" (`Button`, ...)
  - Un composant "conteneur" (`Panel`, `Applet`, `Frame`, ...)
- La classe `Event` définit un ensemble de constantes utiles à la gestion du clavier et de la souris
  - heure, position, état des touches de fonctions, ...
- Pour traiter l'événement, il suffit de redéfinir la méthode correspondant à son type, celle-ci devant être rattachée à l'objet recevant l'événement.

Gestion des évènements dans une fenêtre

- Les événements utilisateurs sont gérés par plusieurs interfaces `EventListener`.
- Les interfaces `EventListener` permettent à un composant de générer des événements utilisateurs.
- Une classe doit contenir une interface auditeur pour chaque type de composant :
  - `ActionListener` : clic de souris ou enfoncement de la touche `Enter`.
  - `ItemListener` : utilisation d'une liste ou d'une case à cocher.
  - `MouseMotionListener` : evénément de souris.
  - `MouseListener` : clic de souris.
  - `WindowListener` : événement de fenêtre.

Utilisation d'une interface EventListener

1. Importer les classes de `java.awt.event` ;
2. La classe doit implanter les interfaces d'écoute

```
public class MaFenetre extends Frame implements ActionListener,
      MouseListener { ...}
```

3. Appel à la méthode `addXXX()` pour enregistrer l'objet qui gerera les évenements `XXX` du composant

```
Button b = new Button("bouton"); b.addActionListener(this);
```

4. Implémenter les méthodes déclarées dans les interfaces ;
   - Pour identifier le composant qui a généré l'evénement utiliser `getActionCommand()` qui renvoie une `String`
   - `getSource()` renvoie l'objet qui a généré l'événement. On préférera cette méthode.

Une première fenêtre avec événements

Utilisation de l'interface `WindowListener`

- Une fenêtre qui n'est pas contenue dans une autre (cadres) dérive de la classe `Frame` ;
- Les cadres sont des exemples de conteneurs ;
- Un conteneur peut contenir des éléments d'interface (boutons, champs de texte ...) ;
- La gestion de la fenêtre se fait par des "écouteurs" d'événements listener ;
- La classe doit implémenter l'interface `WindowListener` :
  - `public void windowClosing(WindowEvent e) ;`
  - `public void windowClosed(WindowEvent e) ;`
  - `public void windowDeiconified(WindowEvent e) ;`
  - `public void windowIconified(WindowEvent e) ;`
  - `public void windowActivated(WindowEvent e) ;`
  - `public void windowDeactivated(WindowEvent e) ;`
  - `public void windowOpened(WindowEvent e) ;`
- On ajoute le listener à la fenêtre par `addWindowListener()`.

Une première fenêtre, le code

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre extends Frame implements WindowListener {
  public Fenetre() {
    setSize(400, 300);
  }
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent event) {}
  public void windowDeiconified(WindowEvent event) {}
  public void windowIconified(WindowEvent event) {}
  public void windowActivated(WindowEvent event) {}
  public void windowDeactivated(WindowEvent event) {}
  public void windowOpened(WindowEvent event) {}

  public static void main(String arg[]) {
    Fenetre Test = new Fenetre();
    Test.setTitle("ma fenetre JAVA");
    Test.setVisible(true);
    Test.addWindowListener(Test);
  }
}
```

Interface `ItemListener`

- Permet de réagir à la sélection de cases à cocher et de liste d'options.
  - Pour qu'un composant genère des évenements, il faut utiliser la méthode `addItemListener()`
  - Evenements reçus par `itemStateChanged()`, attend un objet de type `ItemEvent` en argument ;
  - Pour déterminer si une case à cocher est sélectionnée ou inactive, méthode `getStateChange()`, constantes `ItemEvent.SELECTED` ou `ItemEvent.DESELECTED`
- Exemple.

Interface `TextListener`

- Permet de réagir aux modifications de la zone de saisie ou du texte.
  - `addTextListener()` permet à un composant de texte de générer des événements utilisateur ;
  - `TextValueChanged()` reçoit les évenements.
- Exemple.

Interface `MouseMotionListener`

- Permet de réagir aux changements de position de la souris.
  - `addMouseMotionListener()` permet de gérer les mouvements de la souris ;
  - Les méthodes `mouseDragged()` et `mouseMoved()` reçoivent les évenements.
- Exemple.

Interface MouseListener

- Permet de réagir aux clics de la souris. Les méthodes sont :
  - `public void mouseClicked(MouseEvent e)`
  - `public void mousePressed(MouseEvent e)` : cliqué sur un composant ;
  - `public void mouseReleased(MouseEvent e)` : relaché sur un composant ;
  - `public void mouseEntered(MouseEvent e)` : entre sur composant ;
  - `public void mouseExited(MouseEvent e)` : quitte un composant ·
- Exemple, mauvaise version, pourquoi ?
- Exemple, une meilleure version, pourquoi ?

Les adaptateurs

- Facilite l'implémentation des interfaces ;
- Supposons qu'une classe doivent implémenter `MouseListener` il faut définir 5 méthodes ;
  - Pb toutes les méthodes ne sont pas obligatoirement utilisées ;
  - $\Rightarrow$ on utilise un adaptateur
  - La classe `MouseAdapter` fournit une implémentation par défaut de `MouseListener`
- Exemple

Utilisation d'une classe `Adapter`

- Il suffit de redéfinir la ou les méthodes qui contiendront du code pour traiter les événements concernés ;
- Par défaut, les différentes méthodes définies dans l'`Adapter` ne font rien ;
- La nouvelle classe ainsi définie doit être passée en paramètre à la méthode addXXXListener() au lieu de `this` qui indiquait que la classe répondait elle même à l'événement.
- Il n'existe une classe Adapter que pour les interfaces qui possédent plusieurs méthodes.

Implémentation des Listeners

Une classe implémentant elle-même le listener

```
import java.awt.*;
import java.awt.event.*;
public class Fenetre1 extends Frame implements WindowListener {
 public Fenetre1(String title) {
  super(title);
  this.addWindowListener(this); }
 public static void main(java.lang.String[] args) {
  try {
   Fenetre1 tf = new Fenetre1("Fenetre implémentant elle même le listener");
   tf.setSize(500,100); tf.pack(); tf.setVisible(true);
  } catch (Throwable e) {
     System.err.println("Erreur");
     e.printStackTrace(System.out); }
 }
    // Implémentation de WindowListener
 public void windowActivated(java.awt.event.WindowEvent e) {}
 public void windowClosed(java.awt.event.WindowEvent e) {}
 public void windowClosing(java.awt.event.WindowEvent e) {
  System.exit(0); }
 public void windowDeactivated(java.awt.event.WindowEvent e) {}
 public void windowDeiconified(java.awt.event.WindowEvent e) {}
 public void windowIconified(java.awt.event.WindowEvent e) {}
 public void windowOpened(java.awt.event.WindowEvent e) {}
}
```

Implémentation des Listeners

Une classe indépendante implémentant le listener

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre2 extends Frame {
 public Fenetre2(String title) {
  super(title);
  gestEvt ge = new gestEvt();
  addWindowListener(ge); }

 public static void main(String[] args) {
  try {
   Fenetre2 tf = new Fenetre2("Classe indépendante implémentant le listener");
   tf.setSize(500,100); tf.pack(); tf.setVisible(true);} catch (Throwable e)
    { System.err.println("Erreur"); e.printStackTrace(System.out); }
 }
}

class gestEvt implements WindowListener {
 public void windowActivated(java.awt.event.WindowEvent e) {}
 public void windowClosed(java.awt.event.WindowEvent e) {}
 public void windowClosing(java.awt.event.WindowEvent e) { System.exit(0); }
 public void windowDeactivated(java.awt.event.WindowEvent e) {}
 public void windowDeiconified(java.awt.event.WindowEvent e) {}
 public void windowIconified(java.awt.event.WindowEvent e) {}
 public void windowOpened(java.awt.event.WindowEvent e) {}
}
```

Implémentation des Listeners

Une classe interne implémentant le listener

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre3 extends Frame {
 class gestEvt implements WindowListener {
  public void windowActivated(java.awt.event.WindowEvent e) {}
  public void windowClosed(java.awt.event.WindowEvent e) {}
  public void windowClosing(java.awt.event.WindowEvent e) { System.exit(0); }
  public void windowDeactivated(java.awt.event.WindowEvent e) {}
  public void windowDeiconified(java.awt.event.WindowEvent e) {}
  public void windowIconified(java.awt.event.WindowEvent e) {}
  public void windowOpened(java.awt.event.WindowEvent e) {}
 }

 private gestEvt ge = new Fenetre3.gestEvt();

 public Fenetre3(String title) {
  super(title);
  addWindowListener(ge); }

 public static void main(String[] args) {
  try {
   Fenetre3 tf = new Fenetre3("Classe interne implémetant le listener");
   tf.setSize(500,100); tf.pack(); tf.setVisible(true);} catch (Throwable e)
    { System.err.println("Erreur"); e.printStackTrace(System.out); }
 }
}
```

Utilisation des Adapters

Le plus souvent : Une classe interne anonyme

```
import java.awt.*;
import java.awt.event.*;

public class Fenetre4 extends Frame {
    public Fenetre4(String title) {
  super(title);
  addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
          System.exit(0);
            }
        }
        );
    }

    public static void main(String[] args) {
  try {
      Fenetre4 tf = new Fenetre4("Classe anonyme");
      tf.setSize(500,100); tf.pack(); tf.setVisible(true);
  } catch (Throwable e)
      { System.err.println("Erreur"); e.printStackTrace(System.out); }
    }
}
```

Les Applets

- **applet** = programme dans une page HTML.
- `Applet` super-classe de toute applet ;
- Exemple d'applet sans paramètre.

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet
      {
  public void paint(Graphics g) {
    g.drawString("Salut tout le monde"
      ,
                  50, 25);
  }
}
```

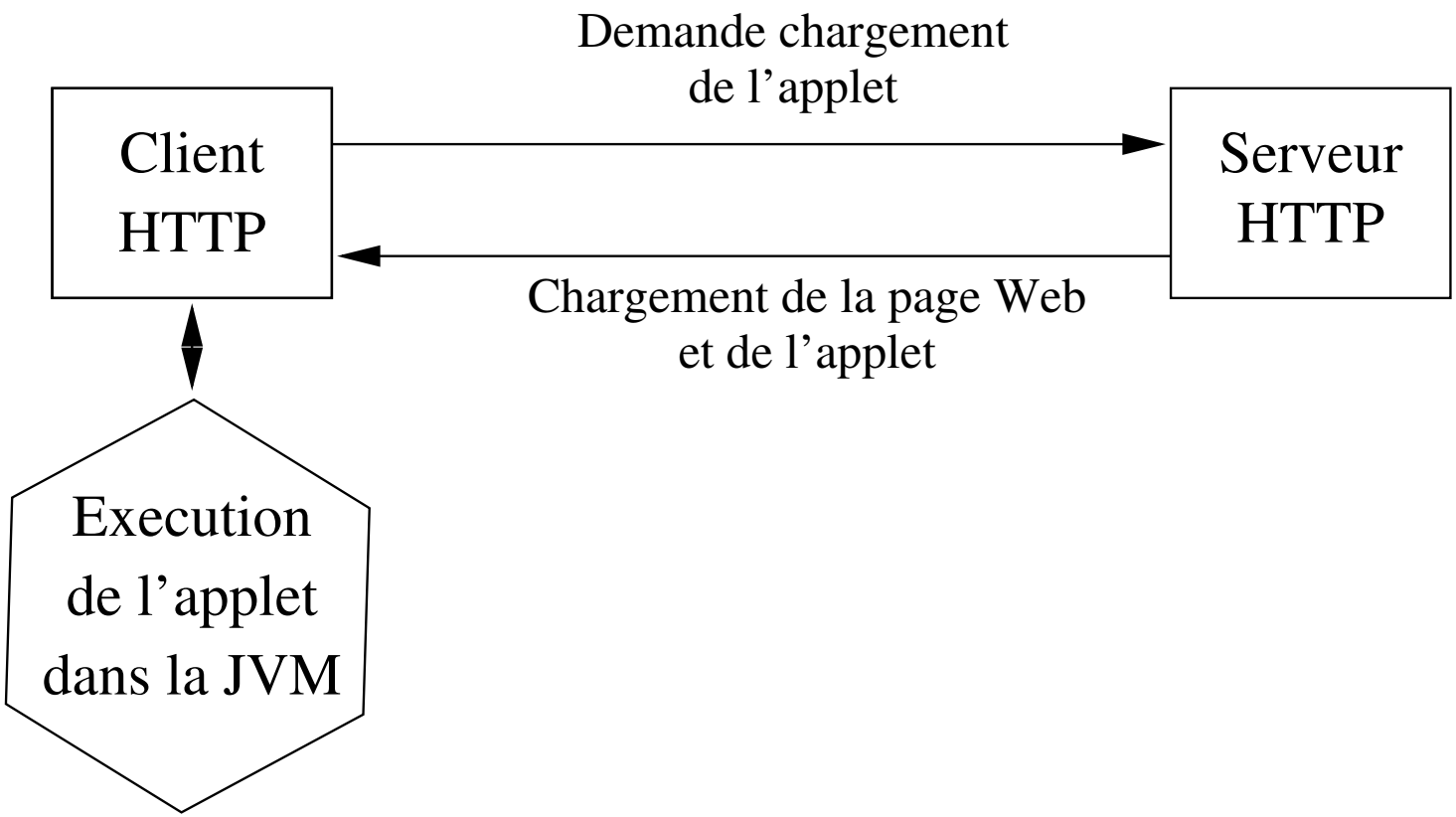
Exemple de page HTML qui appelle l'applet :

```
<html>
  <head>
    <title> Un exemple d'applet </
     title>
  </head>
  <body>
    <h1>Exemple d'une applet</h1>
    <applet code=''HelloWorld.class''
                  width=250  height=25>
    </applet>
  </body>
</html>
```

Applet

- Classe ne possédant pas de méthode `main()` ;
- Son bytecode réside (en général) sur un serveur ;
- Elle est véhiculée dans une page Web (HTML) qui contient son URL, vers un navigateur Web client ;
- Lorsque le navigateur compatible Java (avec sa propre JVM) reçoit cette page HTML, il télécharge (par HTTP) le code de la classe et l'exécute sur le poste client
- Le navigatreur exécute certaines méthodes de l'applet comme `init()`, `start()`, `stop()`, `destroy()`, `paint()`.

Exécution d'une Applet



Comment cela marche ?

- Lorsqu'un navigateur Web compatible Java (avec une JVM) reçoit une page HTML, il télécharge (par HTTP) le code de la classe et l'exécute sur le poste client (ie. invoque ses méthodes `init()` , `start()` , etc) ;
- C'est alors une véritable application qui s'exécute dans la page HTML du navigateur qui peut :
  - Construire et gérer une interface graphique ;
  - Créer de nouveaux threads, ouvrir des connexions réseaux, ...

Des contraintes

- Pour éviter certains ennuis (virus, ...) et pour assurer une exécution sûre sur le poste client, les navigateurs bident le comportement de l'applet.
  - Pas de lecture/écriture de fichiers ;
  - Pas de communication avec d'autres machines que celle depuis laquelle l'applet a été chargée ;
  - Pas d'exécution de programmes (processus) ;
  - Pas de chargement de programmes natifs (binaires, DLL, driver, ...)

Intérêts

- Modifier un programme afin de lui donner un nouveau comportement sans modifier la page HTML ;
- Incorporer une applet dans plusieurs pages HTML et de lui donner des comportements différents selon les paramètres qui lui sont associés ;
- D'autres serveurs peuvent lui faire référence.

Syntaxe de la balise

- `code="UneApplet.class"` (obligatoire), nom de la classe principale ;
- `width=150, height=25` (obligatoire), taille en pixels ;
- `hspace=10, vspace=10`, espace en pixels autour de l'applet ;
- `codebase="codebaseURL"`, URL de base pour l'applet, par défaut c'est le répertoire courant ;
- `name="UneAppletJava"`, pour nommer une applet ;
  - Plusieurs applets nommées peuvent communiquer au sein d'une même page HTML ;
- `align=` (*left* | *right* | *top* | *bottom* `...`) : alignement ;
- `archive="UneArchive.jar"` ou `UneArchive.zip`, fichier archive contenant les classes de l'applet.

Cycle de vie d'une applet

Dans une applet, pas de point d'entrée `main()`, c'est le navigateur qui contrôle la vie et l'activation de l'applet grâce aux quatres méthodes suivantes :

- `public void init()` : appelée après le chargement de l'applet dans la page html ;
- `public void stop()` : appelée à chaque arrêt de l'exécution de l'applet, (l'utilisateur change de page web, ou il iconifie le navigateur ...
- `public void start()` : appelée quand l'applet doit démarrer, après `init()` ou apres un `stop()` lorsque l'utilisateur revient sur la page web contenant l'applet ou lorsqu'il désiconifie le navigateur.
- `public void destroy()` : appelée à la fermeture du navigateur
  - Détruit les ressources allouées pour l'applet.

Écrire une applet

- Dériver la classe `java.applet.Applet`
- Pas de méthode `main()` mais une méthode `init()` ;
- Redéfinir les méthodes héritées pour spécifier le comportement propre de l'applet :
  - `public void init()`
  - `public void start()`
  - `public void stop()`
  - `public void destroy()`
  - `public void paint(Graphics)`

Applet graphique

- L'activité principale est définie dans la méthode `paint()` ;
- La méthode `paint()` est invoquée par le navigateur lors d'événements nécessitant de redessiner la surface occupée par l'applet ;
  - Elle reçoit comme paramètre un objet de la classe `Graphics`
    - `Graphics` decrit l'environnement graphique courant :

```
import java.awt.Graphics;
public class MonApplet extends java.applet.Applet {
 public void paint(Graphics gc) {
  gc.drawLine(50, 25, 100, 25);
 }
}
```

Les méthodes graphiques

- `public void repaint()`
  - Rafraîchissement du programme graphique ;
  - Provoque l'appel à la méthode `paint()` par l'AWT ;
  - Ne pas surcharger !
- `public void update(Graphics g)`
  - Appelée par le navigateur pour redessiner un bout du programme graphique
    - Dessine un rectangle de la couleur du fond sur la zone à redessiner, puis appelle `paint()`
- `public void paint(Graphics g)`
  - Ne doit pas être appelée par le programmeur (`repaint()`)

Tracés dans une applet

Tracés géométriques dans la zone graphique d'une applet grâce aux méthodes de la classe `java.awt.Graphics`.

Méthode	description
`public void draw3DRect(int x, int y, int width, int height, boolean raised)`	trace un rectangle en relief
`public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	trace un arc de cercle
`public abstract void drawLine(int x1, int y1, int x2, int y2)`	trace un segment de droite
`public abstract void drawOval(int x, int y, int width, int heignt)`	trace une ellipse
`public abstract void drawPolygon(int xPoints[], int yPoints[], int nPoints)`	trace un polygone
`public void drawRect(int x, int y, int width, int height)`	trace un rectangle vide
`public abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	trace un rectangle vide à bords arrondis
`public void fill3Drect(int x, int y, int width, int height, boolean raised)`	trace un rectangle plein en relief

Tracés dans une applet

Méthode	description
`public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)`	trace un arc de cercle plein
`public abstract void fillOval(int x, int y, int width, int height)`	trace une ellipse pleine
`public abstract void fillPolygon(int xPoints[], int yPoints[], int nPoints)`	trace un polygone plein
`public abstract void fillRect(int x, int y, int width, int height)`	trace un rectangle plein
`public abstract void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)`	trace un rectangle plein à bords arrondis

Tracés dans une applet

Exemple d'applet contenant des graphiques

```
import java.awt.*;

public class Test extends java.applet.Applet {
  public void paint(Graphics g) {
    int i;
    g.setColor(Color.yellow);
    for (i= 0; i<14; i++)
      g.drawLine(10, 10+16*i, 10+16*i, 218);
    for (i= 0; i<14; i++)
      g.drawLine(10+ 16*i, 10, 218,  10+16*i);
    for (i= 0; i<14; i++)
      g.drawLine(10, 218-16*i, 10+16*i, 10);
    for (i= 0; i<14; i++)
      g.drawLine(10+16*i, 218, 218, 218-16*i);
  }
}
```

Son - Image dans une Applet

Des méthodes sont disponibles pour récupérer des images et des sons :

```
public Image getImage(URL url)
public Image getImage(URL url, String name)
public AudioClip getAudioClip(URL url)
public AudioClip getAudioClip(URL url, String name)
```

Des méthodes de la classe `java.applet.AudioClip` permettent de manipuler les sons ainsi récupérés :

```
public abstract void play()
public abstract void loop()
public abstract void stop()
```

Des méthodes sont également disponibles pour jouer directement les sons :

```
public void play(URL url)
public void play(URL url, String name)
```

Applet avec passage de paramètres

Récupération des paramètres passés à une applet avec :

- `public String getParameter(String name)`
- Ne peut être appelée que dans les méthodes `init()` ou `start()` d'une applet

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld2 extends
      Applet
{
  String e;
  public void init() {
    e=getParameter("message");
  }
  public void paint(Graphics g) {
    g.drawString(e, 50, 25);
  }
}
```

Exemple de page HTML qui appelle l'applet :

```
<html>
  <head>
    <title> Un exemple d'applet avec
            paramètres</title>
  </head>
  <body>
    <applet code="HelloWord2.class"
                width=150 height=25>
    <param name="message" value="
     Bonjour">
    </applet>
  </body>
</html>
```

Applet et application

- On ajoute une classe contenant une méthode `main()` à l'applet ;
- On définit une fenêtre qui recevra l'affichage de l'applet ;
- On appele les méthodes `init()` et `start()` ;
- On affiche la fenêtre.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class AppletApplication extends Applet implements WindowListener {
 public static void main(java.lang.String[] args) {
  AppletApplication applet = new AppletApplication();
  Frame frame = new Frame("Applet");
  frame.addWindowListener(applet);
  frame.add("Center", applet);
  frame.setSize(350, 250); frame.setVisible(true);
  applet.init(); applet.start();
 }

 public void paint(Graphics g) {
  super.paint(g);
  g.drawString("Bonjour", 10, 10);
 }
 public void windowActivated(WindowEvent e) { }
 public void windowClosed(WindowEvent e) { }
 public void windowClosing(WindowEvent e) { System.exit(0); }
 public void windowDeactivated(WindowEvent e) { }
 public void windowDeiconified(WindowEvent e) { }
 public void windowIconified(WindowEvent e) { }
 public void windowOpened(WindowEvent e) { }
}
```

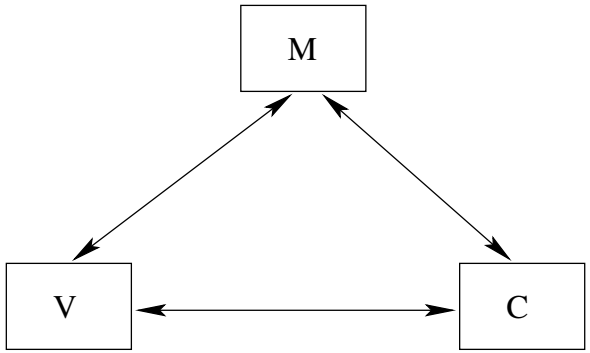
Swing

- Kit de développement d'interface graphique "léger" du JFC (Java Foundation Class), invoqué par le package `javax.swing`
- Simplifie l'utilisation de l'AWT tout en donnant des fonctionnalités avancées pour la création d'interfaces utilisateurs (IHM)
- Portage indépendant du système de fenêtrage - Possibilité de spécifier le "look and feel" qui rapproche le résultat des systèmes fenêtrés connus (Windows, Motif + Métal (Sun) ... limitations dues aux copyrights)

Architecture Swing

- Une application est composée de plusieurs Swing :
  - Un composant de haut niveau ;
  - Plusieurs composants conteneur intermédiaire, ils contiennent d'autre composants ;
  - Des composants atomiques.
- Les composants `Swing` forment un nouvelle hiérarchie parallèle à celle de l'`AWT`.
  - L'ancêtre de cette hiérarchie est le composant `JComponent`.
  - Presque tous ces composants sont écrits en pur Java : ils ne possèdent aucune partie native sauf ceux qui assurent l'interface avec le système d'exploitation : `JApplet`, `JDialog`, `JFrame`, (et `JWindow`).

Le Modèle Vue Controleur

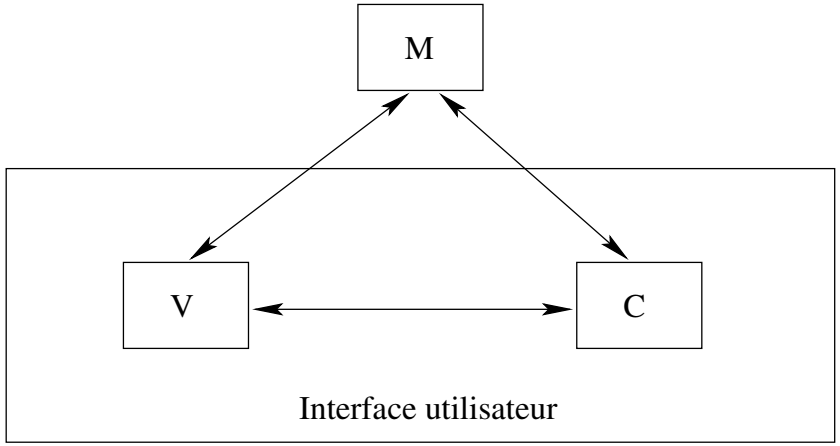


- **Contrôleur** : agit à la demande de l'utilisateur et modifie le modèle ;
- **Modèle** : contient les informations compréhensible par la vue et la notifie des changements ;
- **Vue** : composant affichant le modèle.

Intérêts

- Meilleure abstraction :
  - Facilite le design ;
  - Programmation simplifiée ;
  - Maintenance facilitée, moins de couplage dans le code ;
  - Possibilité d avoir plusieurs vues ou plusieurs contrôleurs ;
  - Possibilité de combiner la vue et le contrôleur ;
  - Jeux d'interfaces et d'événements dédiés à la mise en place simplifiée de ce modèle.

Le Modèle Vue Controleur et Swing



- Le controleur et la vue sont associés pour former l'interface utilisateur ;

Swing et l'AWT

- Swing s'appuie sur l'AWT en particulier pour ce qui concerne :
  - Le placement des composants (`LayoutManager`) ;
  - La gestion des événements (`EventManager`).
- Les classes de Swing sont souvent des classes dérivées de l'AWT (préfixées par J : `JFrame`, `JButton`, `...`) avec des fonctionnalitées d'utilisation simplifiées.
- `javax.swing.JApplet` classe d'applets d'utilisation similaire à `java.applet.Applet` mais qui doit être utilisée si on intègre des composants Swing.

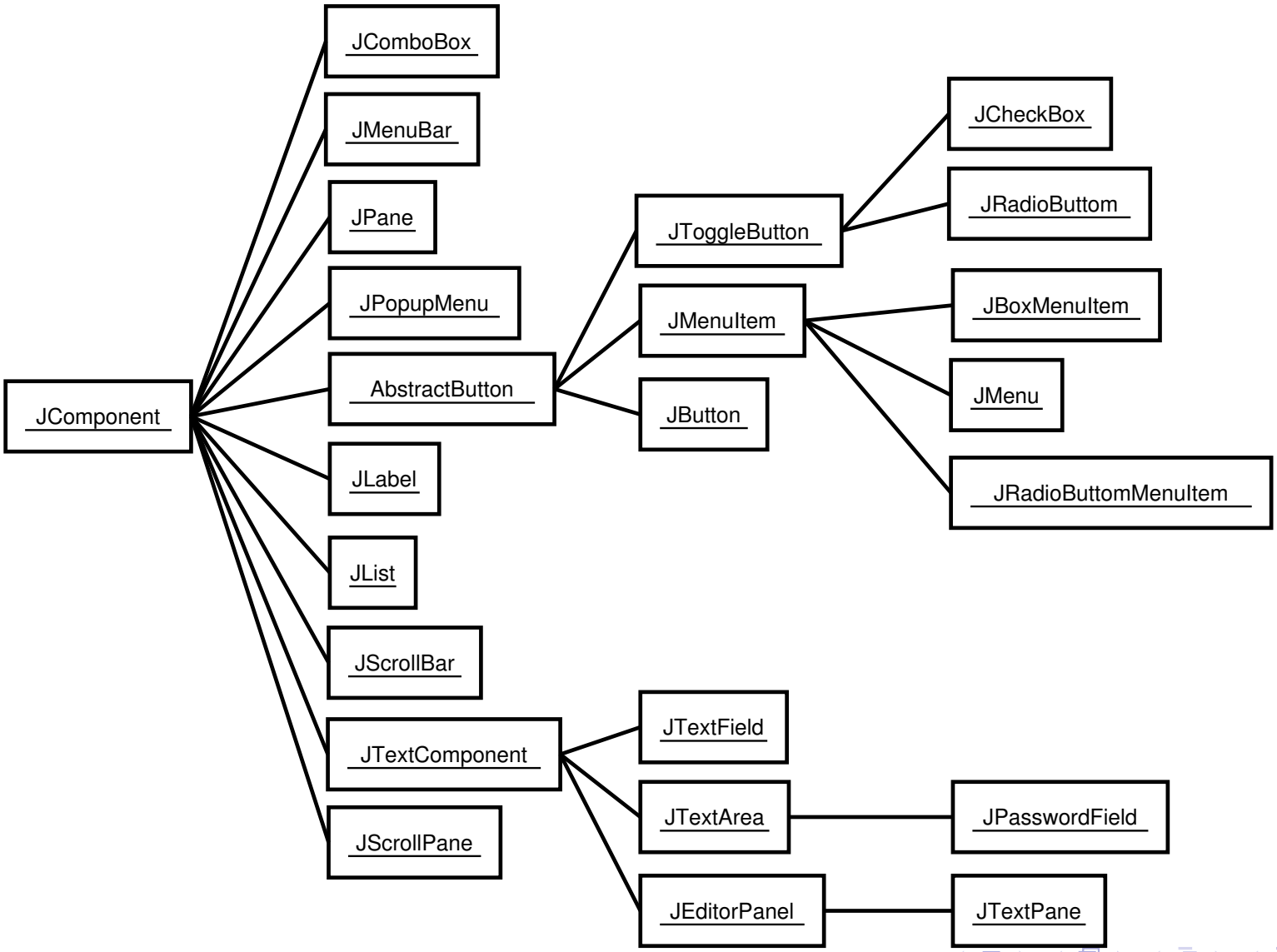
Swing - Exemple de JApplet

On reprend l'applet "HelloWorld"

```
import java.awt.*;

public class JappletHello
  extends javax.swing.JApplet {
    public void paint (Graphics g) {
        g.drawString("Salut tout le monde", 50, 25);
    }
}
```

Les Classes Swing



Le composant JComponent

- Tous les composants Swing héritent de `JComponent` ;
- Les composants ont des "Tool Tips" (Bulles d'aide) ;
- Les composants ont des bordures ;
- Entité graphique la plus abstraite.

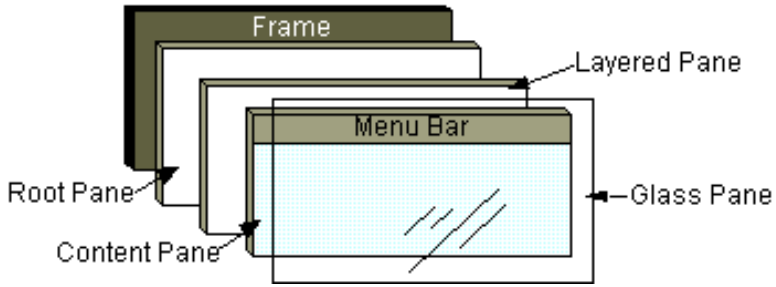
Les composants de haut niveau

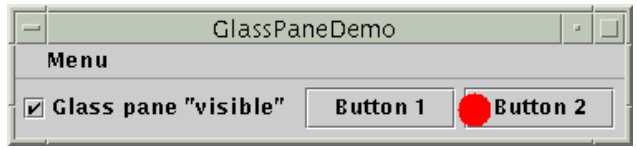
Swing propose 3 composants de haut niveau :

- `JFrame`, `JDialog` et `JApplet` ;
- `JWindow` est aussi de haut-niveau mais "il n'est pas utilisé" ;
- Une application graphique doit avoir un composant de haut niveau comme composant racine (composant qui inclus tous les autres composants).

Les composants de haut niveau

Ils sont structurés en plusieurs couches superposées gérées par le `JRootPane` :

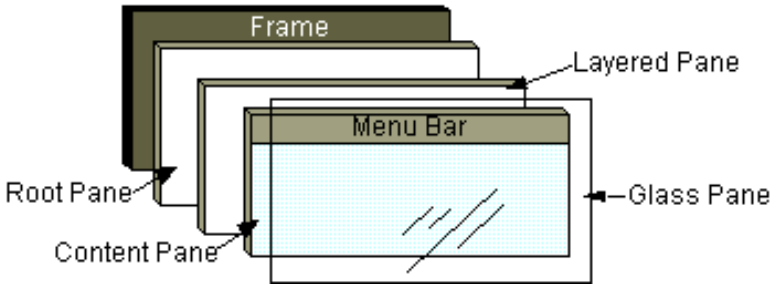


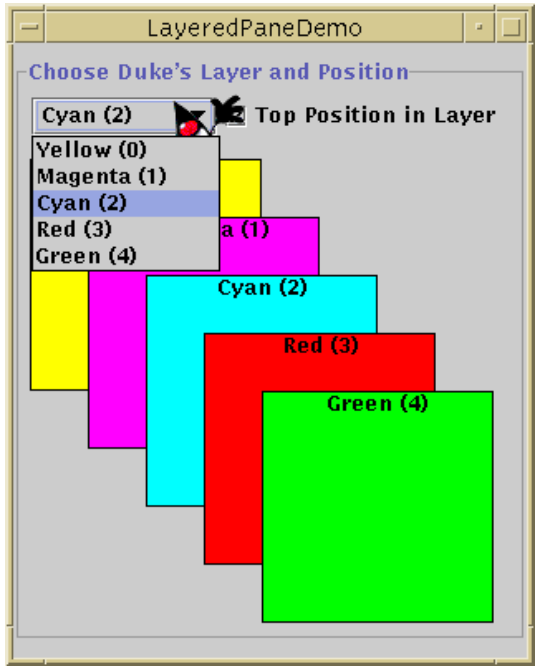


- `glassPane` masquée par défaut, transparent ;

Les composants de haut niveau

Ils sont structurés en plusieurs couches superposées gérées par le `JRootPane` :

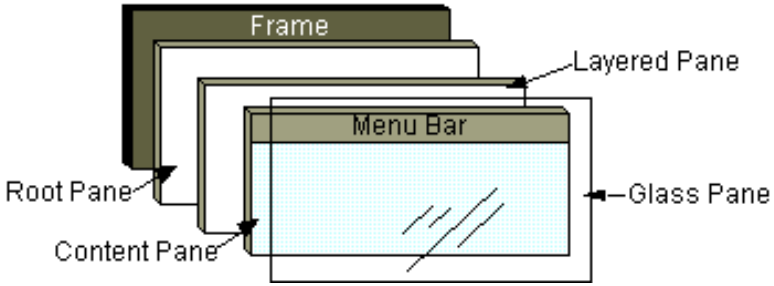


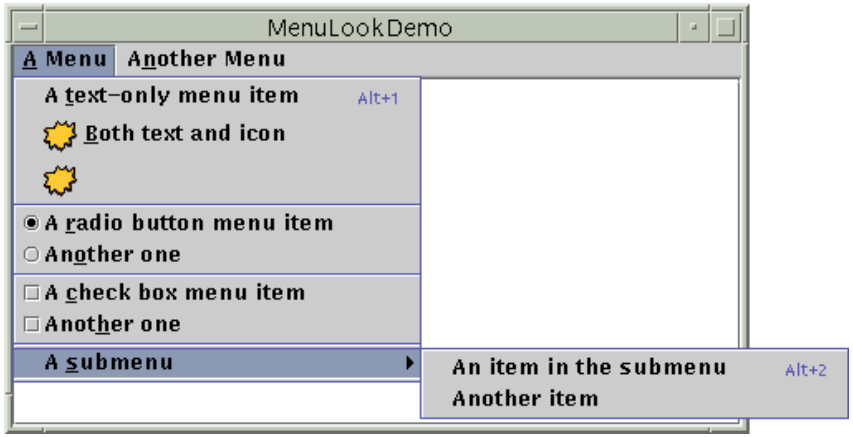


- `glassPane` masquée par défaut, transparent ;
- `layeredPane` gestionnaire

Les composants de haut niveau

Ils sont structurés en plusieurs couches superposées gérées par le `JRootPane` :





- `glassPane` masquée par défaut, transparent ;
- `layeredPane` gestionnaire

Utilisation de cadres JFrame

Utilisation des couches superposées :

1. Utilisation du `JRoot` pour le *look and feel*", p.e.
2. Gestionnaire de mise en page : permet de choisir des modèles de placement de conteneurs de composants (à gérer de manière facultative) ;
3. Gestionnaire de barres de menus : pour placer des menus déroulants (à gérer de manière facultative) ;
4. Couche de contenu / container : tout composant ne peut être placé que dans un container (différence avec AWT), il faut le gérer obligatoirement ;
5. Couche des composants à placer dans le container :

JFrame

- Quelques méthodes :
  - `public JFrame();`
  - `public JFrame(String name);`
  - `public Container getContentPane();`
  - `public void setJMenuBar(JMenuBar menu);`
  - `public void setTitle(String title);`
  - `public void setIconImage(Image image).`

JFrame - un exemple

Le programme qui va suivre gère un `JFrame` :

- On le dimensionne et on lui donne un titre.
- On gère sa fermeture d'une manière simplifiée : on n'implémente pas l'interface `WindowListener` mais on invoque la méthode `addWindowListener` avec un paramètre de type `WindowAdapter` sur lequel on peut ne redéfinir que l'opérateur de fermeture `windowClosing()`.
- On récupère le container du `JFrame` sur lequel on place un composant *volet/panneau* de type `JPanel` qui va permettre l'affichage d'une chaîne de caractères.
- Pour faire cela, on construit une classe dérivant de `JPanel` qui implémente la méthode `paintComponent` remplaçant la méthode `paint()` de l'AWT (à ne plus utiliser directement).

JFrame - un exemple

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class HelloPanel extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString("Bonjour", 75, 100);
}   }
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
}   }
class HelloFrame extends JFrame {
    public HelloFrame() {
        setTitle("HelloInFrame"); setSize(300, 200);
        addWindowListener(new Terminator());
        Container HelloContainer = getContentPane();
        HelloContainer.add(new HelloPanel());
}   }
public class HelloInFrame {
    public static void main(String[] args) {
        JFrame frame = new HelloFrame();
        frame.setVisible(true);
}   }
```

JDialog

- Les dialogues sont des conteneurs de haut niveau (comme `JFrame`).
  - Ils ne sont pas contenus dans d'autres fenêtres, mais dépendent d'une autre fenêtre, si celle-ci est détruite, le `JDialog` l'est aussi.
  - Lorsqu'un cadre est fermé (ou iconifié) les cadres qui en dépendent sont fermés (ou iconifiés).
- Un dialogue peut être :
  - *non modal* : il ne bloque pas les interactions avec les autres fenêtres : pour créer un dialogue non modal il faut utiliser la classe `JDialog`.
  - *modal* : lorsqu'il est actif les interactions avec les autres fenêtres sont bloquées.
- Exemple développé.

Les conteneurs intermédiaires

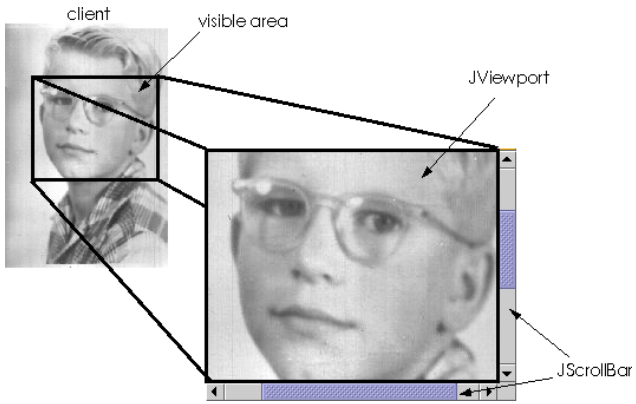
- Swing propose plusieurs conteneurs intermédiaires :
  - `JPanel` ;
  - `JScrollPane` ;
  - `JSplitPane` ;
  - `JTabbedPane` ;
  - `JToolBar` ;
  - .....
- Les conteneurs intermédiaires sont utilisés pour structurer l'application graphique ;
- Le composant de haut niveau contient des composants conteneur intermédiaire ;
- Un conteneur intermédiaire peut contenir d'autres conteneurs intermédiaires.

JPanel

- Utilisation identique aux `Panel` de l'AWT ;
- La classe `JPanel` est un conteneur utilisé pour regrouper et organiser des composants grâce à un gestionnaire de présentation (layout manager) ;
- Le gestionnaire par défaut d'un JPanel est un objet de la classe `FlowLayout`.
- Quelques méthodes de JPanel :
  - `public JPanel();`
  - `public Component add(Component comp);`
  - `public void setLayout(LayoutManager lm);`
- Exemple développé.

JScrollPane

- Gère automatiquement des ascenseurs autour de son composant central qui est un `JViewPort` ;
- Constructeurs principaux :
  - `JScrollPane()` ;
  - `JScrollPane(Component view)` ;
- Une "vue" s'ajoute au `JViewPort`, si elle ne l'est dans le constructeur, par :
  - `scrollPane.getViewPort().add(view)`
- Exemple1 ou encore Exemple 2.



JSplitpane

- Panneau à compartiments, chaque compartiment est ajustable ;
- Panneau à séparation verticale ou horizontale
- Quelques Méthodes :
  - `public JSplitPane(int ori, Component c1, Component c2) ;`
  - `public void setDividerLocation(double pourcentage) ;`
- Exemple détaillé ou encore un autre exemple plus sophistiqué (Berstel).

JTabbedPane

- Un JTabbedPane permet d'avoir des onglets
- Quelques méthodes :
  - `public JTabbedPane();`
  - `public void addTab(String s, Icon i, Component c, String s);`
  - `public Component getSelectedComponent();`
- Exemple détaillé.

JToolBar

- Conteneur général, qui se retaille et peut être déplacé ;
- Utilise un `BoxLayout` horizontal, et ses composants peuvent donc être espacés ou groupés ;
- Les principales méthodes :
  - `JToolbar()` un constructeur ;
  - Ajouter des composants :
    - `JButton add(Action)` ;
    - `Component add(Component)` ;
    - `void addSeparator()`.
  - `ToolBar` flottante :
    - `void setFloatable(boolean)` ;
    - `boolean isFloatable()`.
- Exemple détaillé.

Conteneurs intermédiaires spécialisés

- Les conteneurs intermédiaires spécialisés sont des conteneurs qui offrent des propriétés particulières aux composants qu'ils accueillent.
  - `JRootPane`
  - `JLayeredPane`
  - `JInternalFrame`

JInternalFrame

- Permet d'avoir des petites fenêtres dans une fenêtre ;
- Ressemble très fortement à une `JFrame` mais ce n'est pas un container haut-niveau ;
- Exemple.

Les composants atomiques

- Un composant atomique est considéré comme étant une entité unique.
- Java propose beaucoup de composants atomiques :
  - `JButton, JCheckBox,`
  - `JComboBox`
  - `JList, JMenu`
  - `JTextField, JTextArea, JLabel`
  - `JFileChooser, JColorChooser, ...`

Les boutons

- Java propose différent type de boutons :
  - `JButton` bouton classique ;
  - `JCheckBox` pour les cases à cocher ;
  - `JRadioButton` pour un ensemble de boutons ;
  - `JMenutItem` pour un bouton dans un menu ;
  - `JCheckBoxMenuItem` ;
  - `JRadioButtonMenuItem` ;
  - `JToggleButton` classe mère de `JCheckBox` et `JRadio` ;

JButton

- Un Bouton est un composant essentiel de l'interface utilisateur. Il sert à déclencher une action par une simple pression sur lui.
- Un bouton délivre un `ActionEvent` quand on clique dessus, en plus des évènements de ses superclasses.
- Les méthodes :
  - Les constructeurs :
    ```
    JButton(String, Icon);
    JButton(String);
    JButton(Icon);
    JButton();
    ```
  - Texte du bouton :
    ```
    void setText(String);
    String getText();
    ```
  - Activer / désactiver un bouton :
    ```
    void setEnable(boolean);
    ```
  - Bouton actif au passage de la souris :
    ```
    void setRolloverEnabled(boolean);
    boolean getRolloverEnabled();
    void setRolloverIcon(Icon);
    Icon getRolloverIcon();
    void
    setRolloverSelectedIcon(Icon);
    Icon getRolloverSelectedIcon();
    ```
  - Icône du bouton :
    ```
    void setIcon(Icon);
    Icon getIcon();
    void setDisabledIcon(Icon);
    Icon getDisabledIcon();
    void setPressedIcon(Icon);
    Icon getPressedIcon();
    ```
- Exemple.

JCheckBox

- Cases à cocher (JCheckbox) sur lesquelles on peut cliquer ;
- Elles peuvent prendre deux valeurs : Activée / Désactivée (`Selected/Deselected`) ;
- Les méthodes (cf `JButton`) ;
- Exemple.
- On peut utiliser la classe `ButtonGroup` qui permet de gérer un ensemble de boutons en garantissant qu'un seul bouton du groupe sera sélectionné.
- Exemple.

JRadioButton

- Les boutons radio ressemblent aux cases à cocher sauf qu'en génèral un seul peut être sélectionné ;
- La création de boutons radios passe par la création d'un groupe de cases `ButtonGroup` ;
- Exemple.

JComboBox

- Une liste déroulante est une liste qui apparaît pour permettre à un utilisateur de sélectionner un élément et un seul. Après la sélection la liste se referme.
- Quelques méthodes :
  - Les constructeurs : `JComboBox()`, `JComboBox(ComboBoxModel)`, `JComboBox(Object[])` et `JComboBox(Vector)`.
  - Gestion des items de la liste : `void addItem(Object)`, `void insertItemAt(Object, int)`, `Object getItemAt(int)`, `Object getSelectedItem()`, `void removeAllItems()`, `void removeItemAt(int)`, `void removeItem(Object)` et `int getItemCount()`.
- Exemple.

Jlist

- Propose plusieurs éléments rangés en colonne ;
- Peut proposer une sélection simple ou multiple ;
- Implémentent l'interface `Scrollable` et peuvent donc être placées dans un conteneur `JScrollPane` avec des ascenseurs.
- Les méthodes :
  - Constructeurs `JList(ListModel)`, `JList(Object[])` et `JList(Vector)`.
  - Quatre méthodes pour récupérer le ou les valeurs sélectionnées `Object getSelectedValue()`, `Object[] getSelectedValues()`, `int getSelectedIndex()` et `int[] getSelectedIndices()`.
- Exemple.

JSlider

- Permettent la saisie graphique d'un nombre ;
- Un JSlider doit contenir les bornes max et min `JSlider(int min ,int max, int value)` ;
- Exemple.

JTextField

- Un champ texte est un composant de l'interface utilisateur destiné à la saisie de données alphanumériques ;
- Constructeurs : `JTextField()`, `JTextField(String)`, `JTextField(String, int)`, `JTextField(int)`, et tt JTextField(Document, String, int).
- Permettre ou non l'édition : `setEditable(boolean)`, `boolean isEditable()` ;
- Gestion de la taille : `void setColumns(int)`, `int getColumns()`, `void setHorizontalAlignment(int)`, `int getHorizontalAlignment()`.
- Exemple.

JLabel

- Permet d'afficher du texte ou une image ;
- Peut contenir plusieurs lignes ;
- Comprend les balises HTML ;
- Exemple.

Les Menus

- Barre de menu `JMenuBar` ;
- on ajoute des :
  - `JMenu` dans la barre ;
  - Choix dans les menus `JMenuItem` ;
  - On ajoute un écouteur pour traiter les événements.
- Exemple.

Fenêtres de dialogues particulières

- `JFileChooser`
  - Permet de naviguer dans le système de fichier ;
  - On peut définir un filtre (ex : `*.gif` à l'aide de la classe `FileFilter`.
  - Exemple.
- `JColorChooser`
  - Selection d'une couleur ;
  - Exemple.

Les arbres `JTree`

- Permet une description hiérarchique des données
- Sept autres classes utilisées :
  - *TreeModel* : contient les données figurant dans l'arbre ;
  - *TreeNode* : implémentation des nœuds et de la structure d'arbre ;
  - *TreeSelectionModel* : contient le ou les nœuds sélectionnés ;
  - `TreePath` : un tel objet contient un chemin (de la racine vers le sommet sélectionné par exemple) ;
  - *TreeCellRenderer* : est appelé pour dessiner un nœud ;
  - *TreeCellEditor* : l'éditeur pour un noeud est éditable ;
  - `TreeUI` : look-and-feel.

TreeModel

- Un arbre est créé à partir d'un *TreeModel*

```
interface TreeModel {...
  public Object getChild(Object parent, int index);
  public Object getRoot();
  public boolean isLeaf(Object node);
  ...
}
```

- Il existe plusieurs modèles de sélection :
  - Sélection d'un seul élément ;
  - Sélection de plusieurs éléments contigus ;
  - Sélection de plusieurs éléments disparates.
- Les classes `DefaultXXX` implantent les interfaces *XXX* ;
- On peut indiquer un `DefaultTreeCellRenderer` pour afficher une cellule de façon particulière ;
- On peut indiquer un `DefaultTreeCellEditor` pour changer la valeur d'une cellule.

Utilisation du modèle MVC

- JTree fournit une vue du modèle
  - Les constructeurs :

```
JTree()
JTree(TreeNode racine)
JTree(TreeNode racine, boolean enfantsPermis)
JTree(TreeModel modele)
JTree(TreeModel modele, boolean enfantsPermis)
```

- Le modèle d'arbre est en deux étapes :

```
interface TreeModel
class DefaultTreeModel implements TreeModel
```

- Le modèle des nœuds :

```
interface TreeNode
interface MutableTreeNode extends TreeNode
class DefaultMutableTreeNode implements MutableTreeNode
```

- Exemple.

Les Tables

- `JTable` affiche des données dans un tableau ;
- `TableModel` régit la gestion des données ;
- On fournit soit :
  - Les données dans un tableau bidimensionnel d'objets `Object[][]` et on utilise le `DefaultTableModel` ;
  - On étend `AbstractTableModel`.
- On peut détecter les selections des utilisateurs *ListSelectionModel*.
- Exemple.

Utilisation du MVC et des filtres

- On définit un modèle de gestion des données

```
class FiltreTriModel extends AbstractTableModel
{
 //....
 public int getRowCount(){return modele.getRowCount();}
 public int getColumnCount(){return modele.getColumnCount();}
 public String getColumnName(int c){
   return modele.getColumnName(c);}
 }
```

- La sortie du modèle par défaut `DefaultTableModel` est transmise en entrée au filtre dont la sortie est le modèle de gestion de données.

```
//...
DefaultTableModel modele =
    new DefaultTableModel(cellules, columnNames);
FiltreTriModel trier = new FiltreTriModel(modele);
JTable table = new JTable(trier);
trier.addEcouteur(table);
//...
```

- Exemple.

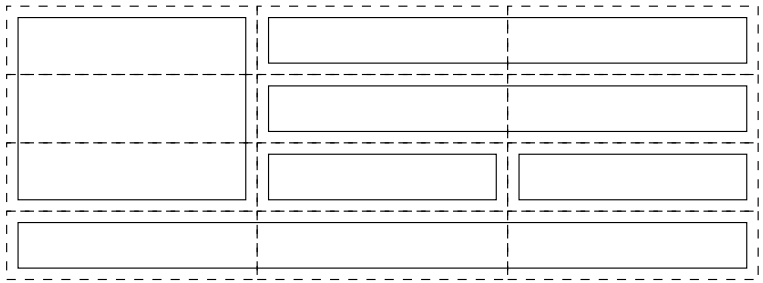
Gestionnaires de mise en page

Idem AWT, ceci est un rappel

Java permet de définir des modèles de mise en page des composants avec la classe `java.awt.LayoutManager`. L'association du gestionnaire de mise en page avec le container se fait par la méthode `setLayout` de l'objet container. Les principaux modèles de mise en page sont :

- `FlowLayout` : modèle par défaut qui positionnne les composants sur une même ligne ou sur la ligne suivante si la ligne courante est remplie.
- `GridLayout(int nl, int nc)` : est un modèle de placement de composants dans un container découpé régulièrement en `nl` lignes et `nc` colonnes.
- `BorderLayout` : modèle de placement des composants en 4 bords (`NORTH`, `SOUTH`, `EAST`, `WEST`) et un centre (`CENTER`).
- `CardLayout` permet l'affichage de composants empilés ; on dispose de méthodes de type `next()`, `previous()` pour les parcourir ;

- `GridBagLayout` est le gestionnaire le plus riche : il étend `GridLayout` en permettant des placements de composants sur des cellules fusionnées (voir figure). On dispose d'objets de contrôle de placement `GridBagConstraints` qui permettent de définir des contraintes de placements pour chaque composant (nombre de cellules occupées, positionnement du coin supérieur gauche, ...)



Gestion des évènements (rappel)

Déjà présenté dans le cadre de l'`AWT`

Modèle observateur-observé mis en place depuis l'API 1.1. Deux types d'objets interviennent :

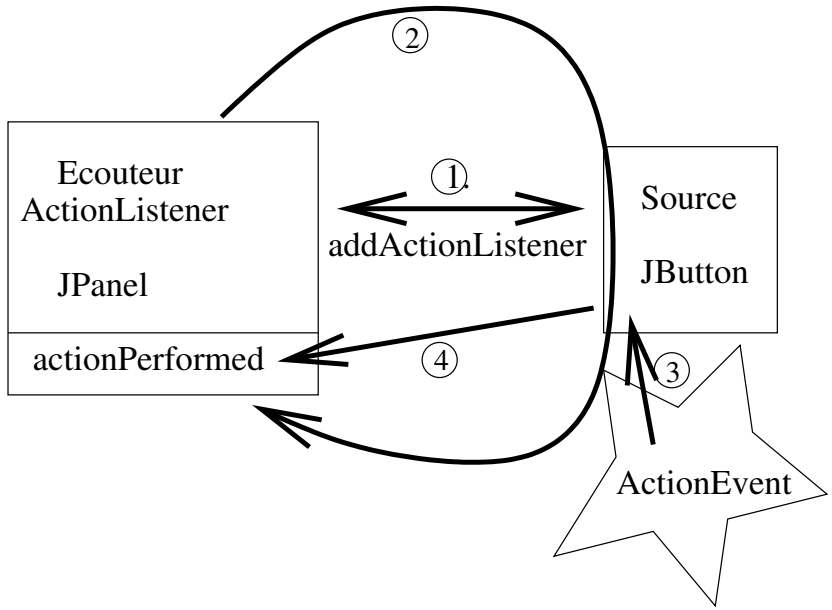
- sources : composants graphiques réactifs à certains types d'évènements ;
- écouteurs objets qui implémentent une interface dérivée de `java.util.EventListener`.

Par exemple, un bouton (`JButton`) peut avoir pour écouteur le volet (`JPanel`) dans lequel il est placé.
Il existe plusieurs types d'écouteurs : `ActionListener`, `FocusListener`, `KeyListener`, `MouseListener`, ...

Gestion des évènements - schéma

Voici la suite de processus engendrée par une action de type `ActionEvent` avec un écouteur de type `ActionListener` :

- La source se lie à un ou des écouteurs (avec la méthode `addActionListener`)
- La source possède, par nature, des méthodes capables de reconnaître certains évènements (clic de souris, p.e.). Lorsque cet évènement (objet de type `ActionEvent`) est détecté, la source envoie l'évènement à l'écouteur qui appelle la méthode spécifique de gestion de l'évènement (`actionPerformed`) et le traite suivant sa nature.



Interface *Action*

- Il arrive souvent qu'une même action (par exemple, quitter l'application, imprimer, ouvrir un fichier, obtenir de l'aide) puisse être déclenchée par différents moyens :
  - choix d'un menu ;
  - clic sur un bouton de la souris ;
  - frappe d'une combinaison de touches au clavier ;
  - etc
- $\Rightarrow$ utilisation de l'interface *Action*.

Interface *Action*

- L'interface Action hérite de `ActionListener` et a les méthodes suivantes :
  - `actionPerformed` (héritée de `ActionListener`) ;
  - `setEnabled` et `isEnabled` indiquent si l'action peut être lancée ou non ;
  - `putValue` et `getValue` permettent d'ajouter des attributs (paire "nom-valeur") à l'action ;
    - Deux attributs prédéfinis : `Action.NAME` et `Action.SMALL_ICON` (utilisés si l'action est associée à un menu ou à une barre d'outils)
  - `{add|remove}PropertyChangeListener` pour, par exemple, notifier un menu associé à une action que l'action est invalidée.

Classes pouvant utiliser *Action*

- Sous-classes de `AbstractButton`, elles ont en particulier un constructeur qui prend en entrée une `Action` :
  - Boutons `JButton` ;
  - Boutons radio `JRadioButton`, y compris dans un `JRadioButtonMenuItem` ;
  - Boîtes à cocher `JCheckBox`, y compris dans un menu `JCheckBoxMenuItem` ;
  - Menu `JMenu` ;
  - Choix de menu `JMenuItem` ;
- Exemple.

Partie 8

Les threads

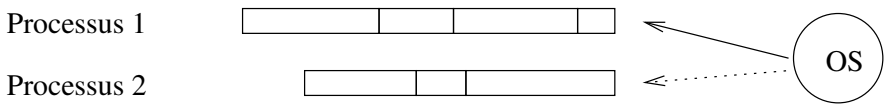
Plan de la partie

Les threads

- Un programme est caractérisé par
  - son code
  - son espace mémoire (données)
- Sur un OS évolué, exécution concurrente possible des processus avec “temps partagé” géré par un scheduler

Si processus indépendants :

- contexte différent à chaque changement de processus ;
- possibilité de partager une mémoire (lourd à gérer) ;
- communications parfois nécessaires.



Une solution plus simple : les threads ou processus légers qui s’exécutent en parallèle mais partagent les données.

Exemple : concurrence de traitement lors de chargements d’images avec les navigateurs Web ou environnement graphique Java.

Création de threads avec Java

Threads gérés dans différents langages. Gestion lourde avec C et simplifiée avec Java qui propose deux solutions :

- Objet héritant de la classe `java.lang.Thread`. Deux objets de cette classe peuvent s'exécuter directement en concurrence.
- Objet implémentant l'interface `java.lang.Runnable`
  - On doit définir une méthode `run()` ;
  - On range ces objets dans des objets Threads (enveloppes) qui s'exécutent de façon concurrente.

Exemples : TestThread1.java et TestThread2.java

Hériter de la classe Thread

```
public class Proc1 extends java.lang.Thread {
 public Proc1() {...} // Le constructeur
 ...

 public void run() {
 ... // Ici ce que fait le processus (boucle infinie)
 }
}
 ...
 Proc1 p1 = new Proc1(); // Création du processus
 p1.start(); // Démarre le processus
             // et exécute p1.run()
```

- Exemple : TestThread1.java

Implémenter l'interface *Runnable*

```
public class Proc2 implements java.lang.Runnable {
 public Proc2() {...} // Constructeur
 ...
 public void run() {
  ... // Ici ce que fait le processus (boucle infinie)
 }
}
...
 Proc2 p = new Proc2();
 Thread p2 = new Thread(p);
 p2.start(); // Démarre un processus qui exécute p.run()
```

- Exemple : TestThread2.java

Que choisir ?

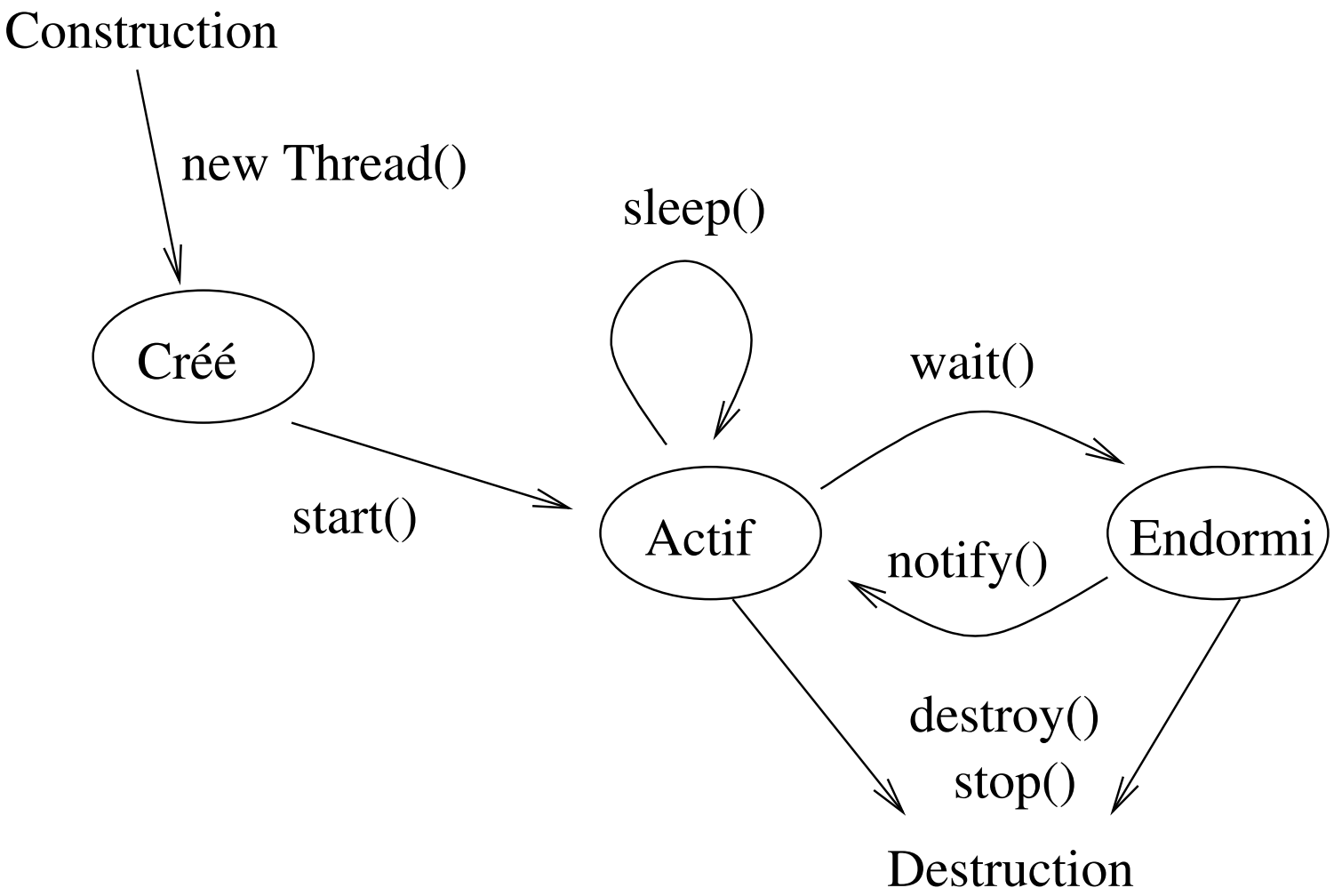
- Hériter de `Thread`
  - Pour paralléliser une classe qui n'hérite pas déjà d'une autre classe (héritage simple) ;
  - cas des applications autonomes.
- Implémenter *Runnable*
  - Lorsqu'une super-classe est imposée ;
  - Cas des applets

```
public class MyThreadApplet
extends Applet implements Runnable {...}
```

Méthodes de la classe *Thread*

Méthode	Rôle
`void start()`	démarre le thread et exécute la méthode `run()`
`static void sleep(long) throws InterruptedException`	Met le thread en attente durant le temps exprimé en millisecondes. Leve une exception de type `InterruptedException` si le thread est réactivé avant la fin du temps.
`void join() throws InterruptedException`	Permet d'attendre la fin d'exécution du thread
`void interrupt()`	Un thread peut se mettre en attente par la méthode `sleep`, ou par l'attente d'une entrée-sortie, ou par `wait` ou `join` Un autre thread peut interrompre cette attente par la méthode `interrupt()`
`int getPriority()`	Renvoie la priorité du thread
`void setPriority(int)`	Fixe la priorité du thread
`static Thread currentThread()`	Placée dans une méthode de n'importe quelle classe, elle retourne l'objet Thread qui contrôle le thread qui exécute cette méthode
`static void yield()`	Passe la main à un autre thread de priorité $\geq$

Les méthodes de gestion des threads



Synchronisation et Exclusion

Lorsque plusieurs threads travaillent sur des mêmes ressources, il est parfois utile ou nécessaire de limiter le parallélisme en assurant qu'un certain objet ne subisse pas en même temps plusieurs séquences d'actions.

Exemple : Dans TestThread3.java, chaque thread affiche des mots caractère par caractère. Les mots sont mélangés. On souhaite conserver le parallélisme du traitement en garantissant que chaque mot entier ne soit pas coupé.

Synchronisation et Exclusion

Construction d'une classe moniteur

On définit une classe `Moniteur` qui contient une méthode `synchronized`.

- Quand cette méthode est appelée dans un thread, elle bloque les autres threads ;
- Les threads doivent partager le même moniteur $\Rightarrow$ attribut statique ;
- Exemple : TestThread4.java

Attente explicite : `wait()` - `notify()`

Les appels aux méthodes synchronisées sont des sections critiques.
Il faut les utiliser avec prudence :

- Elles suppriment le caractère concurrent du traitement $\Rightarrow$ dégradation des performances ;
- Situation d'interblocage.

Attente explicite : `wait()` - `notify()`

Producteur/Consommateur

- Une mémoire tampon est alimentée par un producteur d'objets et utilisée par un consommateur ;
- On synchronise l'accès à cette mémoire (production et consommation non simultanées) ;
- Si le producteur doit déposer un objet alors qu'il en existe déjà un dans le tampon $\Rightarrow$ blocage ;
- Même problème pour un consommateur qui attend le dépot.

Attente explicite : `wait()` - `notify()`

Relachement d'exclusion

- L'attente du consommateur ou du producteur se fait par `wait()` (exception `InterruptedException`) qui relâche l'exclusion ;
- Un processus sort de son endormissement (attente) quand un autre processus exécute `notify()` ou `notifyAll()` (relance **tous** les processus en attente) ;
- Exemple : ProdConsTest.java

Thread de traitement des événements

- Soit le programme suivant :

```
import javax.swing.*;

public class Bonjour extends JFrame {
 public Bonjour() {
  super("Thread et Swing");
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  getContentPane().add(new JLabel("Bonjour à tous!",
                       JLabel.CENTER));
  pack();
  setVisible(true);
 }

 public static void main(String[] args) {
  new Bonjour();
 }
```

- Que constatez vous ? Pourquoi ?

Thread de traitement des événements

- Le programme est constitué de deux threads :
  - Le thread principal de l'application celui qui se crée quand on lance le programme, et dont la tâche est d'appeler la méthode `main()` qui crée un objet de type `Bonjour` et se termine ;
  - L'exécution de ce programme ne devrait donc durer qu'un instant. Pourtant, non !
  - Un deuxième thread a été créé et gère l'interface graphique,
  - Thread de traitement des événements (event-dispatching thread). Seul autorisé à modifier l'interface.

Utilisation du thread de traitement

- Toutes les opérations sur l'interface doivent être réalisées dans un même thread.
- ```
  import javax.swing.*;
  public class Bonjour extends JFrame {
   public creerEtMontrerInterface() {
    super("Thread et Swing");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    getContentPane().add(new JLabel("Bonjour à tous!",
                         JLabel.CENTER));
    pack();
    setVisible(true);
   }

   public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
       public void run() {
         creerEtMontrerInterface();
       });}}}
  ```
- la méthode `SwingUtilities.invokeLater(obj)` prend pour argument un objet implémentant l'interface *Runnable* ;
- On crée une instance d'une classe anonyme implémentant l'interface *Runnable*.
  ```

Partie 9

Les collections

Plan de la partie

Généralités

- Une collection est un objet qui contient d'autres objets
- Par exemple, un tableau est une collection
- Java fournit d'autres types de collections sous la forme de classes et d'interfaces
- Ces classes et interfaces sont dans le paquetage `java.util`

Généricité

- Avant Java 1.5, les collections peuvent contenir n'importe quel type d'objets ;
- Après, on peut indiquer le type des objets contenus dans une collection grâce à la généricité.

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

  On a créé une liste de `String` ;
- La syntaxe pour mettre en œuvre la généricité utilise les symboles < et > pour préciser le ou les types des objets à utiliser.
- Seuls des objets peuvent être utilisés avec la généricité : si un type primitif est utilisé dans les generics, une erreur de type `unexpected type` est générée lors de la compilation.

Généricité

Exemple de programme qui pouvait poser problème :

```
Collection pommes = new ArrayList() ;
pommes.add(new Pomme()) ;
pommes.add(new Pomme()) ;
pommes.add(new Scoubidou()) ;
for(int i=0 ;i<pommes.size() ;i++){
    // erreur exécution à la 3me itération
    Pomme p = (Pomme)pommes.get(i) ;
    p.extraireJus() ;
}
```

Généricité

- Solution générique

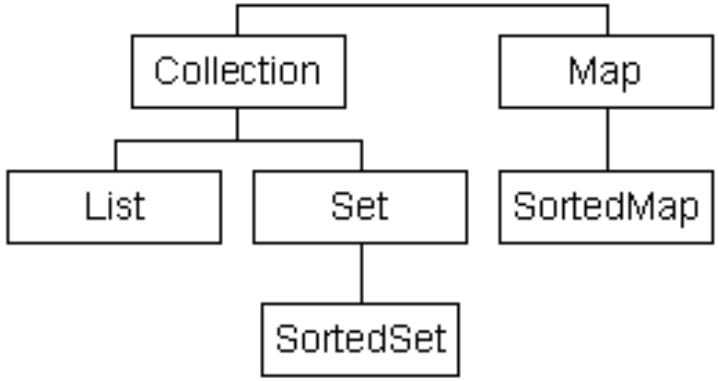
```
Collection<Pomme> pommes = new ArrayList<Pomme>() ;
pommes.add(new Pomme()) ;
pommes.add(new Pomme()) ;
/*pommes.add(new Scoubidou()) ; */
// erreur de compilation
for(int i=0 ;i<pommes.size() ;i++){
    Pomme p = pommes.get(i) ; // casting implicite
    p.extraireJus() ;
}
```

- Le code de la première version est toujours utilisable, mais avertissement de la part du compilateur.

Les Interfaces

- Les interfaces à utiliser par des objets qui gèrent des collections sont :
  - *Collection* : implementée par la plupart des objets qui gèrent des collections ;
  - *Map* : définit des méthodes pour des objets qui gèrent des collections sous la forme clé/valeur ;
  - *Set* : n'autorise pas la gestion des doublons dans l'ensemble ;
  - *List* : autorise la gestion des doublons et un accès direct à un élément ;
  - *SortedSet* : étend *Set* et permet d'ordonner l'ensemble ;
  - *SortedMap* : étend `Map` et permet d'ordonner l'ensemble.
- Certaines méthodes définies dans ces interfaces sont dites optionnelles : leur définition est donc obligatoire mais si l'opération n'est pas supportée alors la méthode doit lever une exception particulière.

Hierarchie des interfaces



- Il existe des classes abstraites qui implémentent un squelettes `AbstractXXX` ex : `AbstractList`.
- et des classes implémentant ces interfaces :

	Set	*List*	*Map*
Tableau redimensionnable		`ArrayList`, `Vector` (1.1)	
Arbre	`TreeSet`		`TreeMap`
Liste chaînée		`LinkedList`	
Collection utilisant une table de hashage		`HashSet`	`HashMap`, `HashTable` (1.1)
Java 1.1		`Stack`	

Interface Collection

Cette interface représente un minimum commun pour les objets qui gèrent des collections

`boolean add(Object)`	ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
`boolean addAll(Collection)`	ajoute à la collection tous les éléments de la collection fournie en paramètre
`void clear()`	supprime tous les éléments de la collection
`boolean contains(Object)`	indique si la collection contient au moins un élément identique à celui fourni en paramètre
`boolean containsAll(Collection)`	indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
`boolean isEmpty()`	indique si la collection est vide
`Iterator iterator()`	renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection
`boolean remove(Object)`	supprime l'élément fourni en paramètre
`boolean removeAll(Collection)`	supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
`int size()`	renvoie le nombre d'éléments contenu dans la collection
`Object[] toArray()`	renvoie d'un tableau d'objets qui contient tous les éléments de la collection

L'interface Iterator

Cette interface définit des méthodes pour des objets capables de parcourir les données d'une collection.

Méthode	Rôle
`boolean hasNext()`	indique si il reste au moins un élément
`Object next()`	renvoie la prochain élément dans la collection
`void remove()`	supprime le dernier élément **parcouru**

```
Iterator iterator = collection.Iterator();
if (iterator.hasNext()) {
     iterator.next();
     iterator.remove();
  }
```

Exemple

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Forme {
  abstract double getAire(); // à définir
  void showAire() {
    System.out.println("aire = "+getAire());}
}
public class Rectangle extends Forme{ ... }
public class Ellipse extends Forme { ... }

public class TestForme {
  public static void main(String[] args) {
   Forme r1 = new Rectangle(6,10);
   Forme r2 = new Rectangle(5,10);
   Forme e = new Ellipse(3,5);
   ArrayList<Forme> liste = new ArrayList<Forme>();
   liste.add(r1);
   liste.add(r2);
   liste.add(1,e);  // on a r1, e, r2
   for (Iterator it = liste.iterator(); it.hasNext();)
     ((Forme) it.next()).showAire();
 }
}
```

Utilisation d'un `for-each`

- Permet le parcours d'une structure de donnée "itérable" ;

```
List<String> noms = new ArrayList<String>();
noms.add("aaaa");
noms.add("bbbb");
noms.add("ccccc");

for (String name: nomss)
   System.out.println(name);
```

Utilisation d'un itérateur

- Permet le parcours d'une structure de donnée "itérable" ;

```
List<String> noms = new ArrayList<String>();
noms.add("aaaa");
noms.add("bbbb");
noms.add("ccccc");

for (Iterator it = noms.iterator(); it.hasNext(); ) {
    String name = (String)it.next();
    System.out.println(name);
 }
```

Utilisation d'un indice

- Utilisation de la méthode get() ;

```
List<String> noms = new ArrayList<String>();
noms.add("aaaa");
noms.add("bbbb");
noms.add("ccccc");

for(int i = 0; i < noms.size(); i++)
    System.out.println(names.get(i));
```

- Attention ne fonctionne pas sur les structures de type ensemble.