# Le langage Java

Jean-Baptiste Yunès

Université Paris 7 - Denis Diderot

Jean-Baptiste.Yunes@liafa.jussieu.fr

# Généralités

La programmation Orientée Objet n'est pas un concept nouveau (smalltalk/simula - années 60).

De nombreux langages orientés objets existent, dont les principaux sont : smalltalk, eiffel, C++ et Java.

Aujourd'hui (presque) tous les langages de programmation sont orientés objets.

# Histoire naturelle

- données structurées (`struct` de C, `record` Pascal, etc)
- données structurées + fonctions (modules)
- modèles, types et instances (objets et classes)
- héritage, polymorphisme (sorte de)
- encapsulation (visibilité)
- implémentation/spécification (types abstraits)
- liaison dynamique/tardive

La POO facilite :

- la séparation de l'implémentation et de la spécification (conception/réalisation)
- la réduction du couplage entre entitées (couplage fort à l'intérieur, couplage faible à l'extérieur),
- la construction d'entités logicielles réutilisables (composants)

# Un exemple : une pile

Une pile est un “objet” sur lequel nous pouvons agir :

- en lui donnant un “objet” à ajouter (“empiler”),
- en lui demandant de retrouver (“dépiler”) un objet précédemment empilé,
- en lui demandant si il y a quelque chose à dépiler

Cette description informelle est une spécification abstraite de ce que nous appelons “pile” (c’est souvent sujet d’infinies discussions)

Une description plus complète devrait indiquer en plus les conditions nécessaires au bon fonctionnement des actions (dépiler dans un pile vide ?), ainsi que les propriétés (axiomes) vérifiées par toute pile (une pile “neuve” est vide, après empilement une pile n’est pas vide, dépiler(empiler(objet)) = objet).

Quelques remarques :

Cette description est indépendante de la représentation d’une pile

réelle (toute pile vérifie la description précédente) : spécification (divers formalismes existent).

On doit pouvoir réaliser des piles comme on le désire (en utilisant des tableaux, des listes) pourvu qu'elles soient conformes à la spécification : implémentation (notion de correction).

L'encapsulation doit assurer que les utilisateurs d'un objet pile quelconque ne pourront atteindre la représentation interne de l'objet : sécurité et couplage faible.

# Un exemple Java : une pile

Voici la définition (abstraite) en Java d'une Pile :

```
package Piles;
public interface Pile {
  public Element pop() throws VideException;
  public void    push(Element e);
  public boolean isEmpty();
  public class VideException extends Exception {}
  public interface Element {}
}
```

La définition du type `Piles.Pile` est une traduction presque directe de la spécification que nous en avions donné.

Certains outils de génie logiciel (AGL) permettent d'ailleurs de générer ce code automatiquement.

On y voit :

- qu'une `Pile` possède une opération `pop()` qui renvoie un `Element`, laquelle pouvant se comporter anormalement en cas de pile vide (exception),
- qu'une `Pile` possède une opération `push()` qui prend en paramètre un `Element`
- que l'on peut tester si une pile est vide (`isEmpty`),
- qu'associé à ces piles nous avons deux autres types : les `Element` et la condition exceptionnelle `VideException`.

Nous pouvons imaginer construire des programmes en utilisant la définition précédente. Par exemple :

```
public class MonApplication {
  public static void application(Pile p,Element e) {
    p.push(e);
    while(! p.isEmpty()) {
      try {
        System.out.println(p.pop());
      } catch (Pile.VideException vide) {
        System.out.println("pop sur pile vide!");
      }
    }
  }
}
```

Le programme ainsi écrit, est compilable normalement (aucun problème de type).

Évidemment son exécution n'est possible que si on lui fournit des objets concrets (l'abstraction n'est utile qu'au raisonnement).

Tous les objets concrets qui satisfont le contrat du type abstrait pourront être utilisés à l'exécution (y compris les plus "triviaux").

Nous avons donc réduit les dépendances entre les entités logicielles. Ce qui permet de les construire séparément.

Ce mécanisme ne donne de bons résultats que si nous avons su définir les "bons" types dès le départ.

Voici la définition (concrète) en Java d'une Pile :

```
package fr.jussieu.liafa.Pile;
import java.util.Vector;
public class PileVecteur implements Pile {
  private Vector t;
  public PileVecteur() { t = new Vector(); }
  public Pile.Element pop() throws Pile.VideException {
    if (isEmpty()) throw new Pile.VideException();
    Pile.Element e = (Pile.Element)(t.lastElement());
    t.removeElementAt(t.size()-1);
    return e;
  }
  public void push(Pile.Element e) { t.add(e); }
  public boolean isEmpty() { return t.isEmpty(); }
}
```

Nous aurions pu définir n'importe quel autre type concret (par exemple une pile utilisant une liste chaînée), pourvu qu'il satisfasse notre définition.

Nous n'avons plus qu'à construire des objets concrets et les donner entrée de notre application :

```
import Piles.*;
class ElementConcret extends Pile.Element {}
public class ProgrammePrincipal {
  public static void main(String argv[]) {
    PileVecteur p = new PileVecteur();
    ElementConcret e = new ElementConcret();
    MonApplication.application(p,e);
  }
}
```

# Les objets

Un objet est unique : il est identifié par un numéro de série appelé identificateur.

Un objet est l'union :
– d'un état (données)
– d'un comportement (actions)
Il y a interaction entre état et comportement.

Exemple d'objet : la montre que je porte (certainement identifiée par un numéro fabricant).

# Les interfaces

Une interface correspond à un type abstrait d'objets : c'est la partie visible du comportement d'un objet.

Une interface représente un ensemble d'objets.

En Java, une interface définit un type pour le système de types du langage.

Exemple d'interface : quelque chose qui donne l'heure.

# Les classes

Une classe est une abstraction de réalisation : c'est un modèle d'objets. On parle parfois de prototype. C'est un moule de construction : un objet est instance d'une classe.

Une classe est construite par factorisation de caractéristiques d'objets (classification : généralisation/spécialisation).

En Java, une classe définit aussi un type pour le système de types.

Exemple de classe : les montres du même modèle que celle que je porte.

# Méthode objet

Contrairement à ce que l'on imagine parfois, une méthode d'analyse orientée objet s'intéresse dans l'ordre :

1. à identifier les objets du problème,
2. à identifier les relations entre objets,
3. à effectuer une classification des objets,
4. à abstraire les relations sur les classes,
5. à déterminer les types abstraits.

La formalisation (par exemple UML) de cette analyse conduit à générer facilement les définitions des types et classes.

Reste à construire les éléments fonctionnels de l'application, ce qui peut s'effectuer avec une certaine garantie (typage).

Reste ensuite à définir des types concrets prototypes permettant de tester l'application.

Les objets seront, eux, créés dynamiquement à l'exécution.

Cette façon de procéder permet normalement de détecter les incohérences graves relativement tôt.

Rappelons simplement que ce qui coûte le plus cher dans le cycle de vie d'une application c'est : l'erreur de conception ! Autant s'en affranchir le plus et le plus tôt possible.

# Généralisation/Spécialisation

La classification conduit très naturellement à effectuer des cascades de factorisation. Ce qui mène à construire un arbre de classification.

Par exemple (extrait) : mammifères → carnivores → canidés → chiens/loups

Dans ce cas, on exhibe des relations entre les ensembles d'objets. Les relations en question sont celles de généralisation/spécialisation qui sont caractérisées par "est-un" ou "est-une-sorte-de".

Cette relation est dirigée et pour une classe donnée on parle de super-classes ou sous-classes (directes ou non).

Cette classification par étage permet de construire différents niveaux de discours.

Attention : il ne faut pas confondre généralisation/spécialisation et héritage/délégation.

La généralisation/spécialisation appartient au domaine de l'analyse (relation entre classes ou types).

L'héritage ou la délégation sont des techniques de programmation qui permettent de réaliser la généralisation/spécialisation.

# Polymorphisme

Le polymorphisme fait référence à l'observation suivante : un objet est instance d'une classe donnée. Cette classe appartient à un arbre de classification et possède donc (possiblement) une cascade de super-classes. Ce qui permet de restreindre la vision de cet objet lorsqu'on le considère comme instance d'une super-classe. L'objet prend alors différentes formes (polymorphisme).

On distingue généralement trois polymorphismes :

- ad-hoc
- inclusion
- paramétrique

# Polymorphisme ad-hoc

C'est un polymorphisme syntaxique qui n'apporte rien de fondamental, mais rend la vie du programmeur plus agréable.

On le rencontre lorsque l'on parle de surcharge (plusieurs fonctions portant le même nom).

Par exemple :

```
int add(int a,int b) { return a+b; }
int add(int a,int b,int c) { return a+b+c; }
float add(float a,float b) { return a+b; }
```

# Polymorphisme par inclusion

C'est un polymorphisme universel.

Il utilise la relation d'inclusion de la généralisation/spécialisation.

Soit une classe $\mathbb{A}$. On peut écrire un programme en termes de cette classe (construire un discours en raisonnant sur $\mathbb{A}$). Alors on doit pouvoir exécuter ce programme normalement en utilisant n'importe quelle instance de $\mathbb{A}$ (le discours est valable pour n'importe quel $a \in \mathbb{A}$). Et donc en particulier pour toute instance d'une sous-classe $\mathbb{B}$ de $\mathbb{A}$.

En effet les objets instances de $\mathbb{B}$ sont conformes vis à vis de la spécification (plus générale) de $\mathbb{A}$, puisque $\mathbb{A}$ est une généralisation de $\mathbb{B}$.

Le terme inclusion provient du fait qu'il existe un ordre sur les types (ici $\mathbb{B} \subset \mathbb{A}$).

Par exemple :

```
void donneAManger(Carnivore c,Viande v) {
  c.metsdanssageule(v);
  c.mastique();
  c.avale();
  c.digere();
}
main() {
  Chat mimine; Chien rex; SteakHache s1, s2;

  donneAManger(mimine,s1);
  donneAManger(rex,s2);
}
```

# Polymorphisme paramétrique

C'est un polymorphisme universel parfois appelé généricité.

Il permet le paramétrage d'un programme sur les types.

Et cette fois c'est la fonction qui sera instanciée (d'une certaine manière) sur les types effectifs à l'exécution.

Par exemple :

```
boolean egal<generic T>(T a,T b) {
  return a == b;
}
main() {
  int a,b; double c[10],d[10];

  egal(a,b);
  egal(c,d);
}
```

# objets, interface et classe

Pour résumer très brièvement :

La montre (objet) que je porte actuellement au poignet a été construite sur le modèle (classe) défini par le fabricant qui s'occupe de fabriquer des choses qui donnent l'heure (interface).

# Java

Java est une marque commerciale détenue par la compagnie Sun microsystems.

Le nom Java recouvre différentes choses :

- un langage de programmation (JLS)
- une machine (JVM)
- un environnement d'exécution (JRE)
- un environnement de développement (SDK)

# Caractéristiques principales

Les caractéristiques souvent citées de Java sont :

- simplicité
- orientation objet
- distribution de l'exécution
- portabilité du code machine
- dynamique
- richesse des packages

# Produits, Documentations

On trouvera à partir de l'URL `http://java.sun.com/` :

- le Java 2 Platform, Standard Edition (J2SE) version 1.4.2 SDK (contenant l'ensemble complet de logiciels permettant de développer en Java) : les plateformes supportées par Sun sont : Solaris SPARC/x86, Linux x86, Microsoft Windows,
- la documentation des diverses API,
- la spécification du langage (JLS),
- la spécification de la machine (JVM)
- un tutorial
- ...

# Les outils

Les applications fournies sont (entre autres) :

- `javac` : le compilateur
- `java` : la machine virtuelle
- `javadoc` : le générateur de documentation
- `appletviewer` : la machine d'exécution d'applets
- `jar` : l'archiveur
- `jdb` : le débogueur

# HelloWorld

Comme attendu, voici un premier exemple de programme Java :

```
public class HelloWorld {
  public static void main(String [] argv) {
    System.out.println("Hello world!");
  }
}
```

Tout d'abord : il est impératif qu'un tel code source soit contenu dans un fichier de nom `HelloWorld.java` (il faut respecter la casse et utiliser le nom de la classe publique définie).

Pour compiler : `javac HelloWorld.java`

En réponse, le compilateur fabrique un fichier de nom `HelloWorld.class`.

Pour exécuter ce code : `java HelloWorld`, et la magie opère.

# Bytecode

La compilation d'un code source Java créé du bytecode Java (JVM).

Le bytecode Java est totalement portable : aucune recompilation n'est nécessaire.

De plus certaines propriétés de sûreté du bytecode sont vérifiées au moment du chargement de celui-ci par la machine virtuelle (Java Verifier).

Il existe pour plus de sécurité un mécanisme de signature et contrôle d'accès du bytecode (Java Security API), afin de contrôler la provenance ou limiter les accès aux ressources pour un code obtenu par le réseau.

# Mots clés

Comme tout langage de programmation, Java possède une liste de les mots clés :

| | | | | |
|---|---|---|---|---|
| `abstract` | `default` | `if` | `private` | `this` |
| `boolean` | `do` | `implements` | `protected` | `throw` |
| `break` | `double` | `import` | `public` | `throws` |
| `byte` | `else` | `instanceof` | `return` | `transient` |
| `case` | `extends` | `int` | `short` | `try` |
| `catch` | `final` | `interface` | `static` | `void` |
| `char` | `finally` | `long` | `strictfp` | `volatile` |
| `class` | `float` | `native` | `super` | `while` |
| `const` | `for` | `new` | `switch` | |
| `continue` | `goto` | `package` | `synchronized` | |

# Types primitifs

Il existe aussi des types de données standardisés :

- les entiers relatifs,
- les réels à virgule flottante,
- les booléens,

Pour les entiers relatifs, on dispose des types normalisés suivants :

| Type | Taille | Représentation |
|---|---|---|
| `byte` | 8 bits | signés en complément à 2 |
| `short` | 16 bits | signés en complément à 2 |
| `char` | 16 bits | non signés (Unicode) |
| `int` | 32 bits | signés en complément à 2 |
| `long` | 64 bits | signés en complément à 2 |

Pour les réels à virgule flottante :

| Type | Taille | Représentation |
|---|---|---|
| `float` | 32 bits | IEEE 754 si (`strictfp`), native sinon |
| `double` | 64 bits | IEEE 754 si (`strictfp`), native sinon |

Le type booléen est `boolean` et ne permet de représenter que deux valeurs `true` et `false`. Ce type doit impérativement être le type des expressions conditionnelles (`if`, `while`, etc.)

Ces types numériques ainsi que les opérateurs afférents fournissent des résultats identiques sur toutes les plateformes.

# Types références

Les objets ne sont pas contenus dans des variables. Ils sont dans un espace particulier.

Les objets ont une vie propre : ils sont alloués dynamiquement et détruits lorsque c'est nécessaire. C'est donc la logique du programme qui contrôle leur existence.

Une variable d'un type autre que primitif ne contient pas d'objet, mais l'identité d'un objet. On peut donc considérer qu'une telle variable (appelée référence) est un nom éventuellement associé à un objet.

On accède jamais directement à un objet, mais seulement en utilisant une référence à celui-ci.

Une variable ne référençant pas d'objet contient la référence `null`.

On peut utiliser le polymorphisme d'inclusion sur les références.

Par exemple :

```
class Vehicule {}
class Voiture extends Vehicule {}
...
Vehicule unVehicule;
Voiture  uneVoiture;
...
unVehicule = uneVoiture; // Ok
uneVoiture = unVehicule // Interdit. Message :
// Incompatible type for =.
// Explicit cast needed to convert Vehicule to Voiture
```

# Les tableaux

Attention : en Java les tableaux sont des objets. On ne déclare donc que des références de tableau.

Par exemple :

```
int a []; // Référence sur un tableau d'int
a = new int[5]; // a dénote un tableau de 5 int
a[0] = 3;
```

Il existe une forme Java de déclaration de référence sur tableau :

```
int [] a; // Plus lisible ?!
int [] a, b[]; // C'est quoi b ???
```

# Définition de classe

La définition d'une classe s'effectue par l'utilisation du mot clé `class`.

Une classe peut être définie comme abstraite, ce qui interdira toute instanciation propre de celle-ci. Pour cela il faut utiliser le mot clé `abstract`. Une telle classe n'a évidemment de sens que si des sous-classes sont définies. On utilise généralement une telle construction (partiellement achevée), par opposition à la définition d'une interface (type abstrait), pour réaliser de l'héritage d'implantation.

Une classe peut être définie comme sous-classe d'une autre en utilisant le mot-clé `extends` (qui exprime la spécialisation). En son absence la super-classe `Object` est utilisée.

Donc toute classe Java est sous-classe de la classe `Object` (ce dont on se servira dans la suite) : un seul arbre de types (`Object` étant

le plus général de tous).

Enfin une classe peut être définie en interdisant toute spécialisation (mot clé `final`). Attention, une classe ne peut être à la fois abstraite et finale.

Exemples :

```
abstract class Vehicule { ... }
class Voiture extends Vehicule { ... }
final class Simca1000 extends Voiture { ... }
```

Le bloc peut contenir (entre autres) la définition de variables ou de méthodes. Et on distingue alors :

- les variables d'instance, qui seront propres à chaque instance,
- les variables de classe (utilisation du mot clé `static`), qui seront partagées par toutes les instances,
- les méthodes d'instance, qui seront propres à chaque instance,
- les méthodes de classe (utilisation du mot clé `static`), qui seront partagées par toutes les instances.

Par exemple :

```
class CompteEnBanque {
  static int nombreDeComptes; // compteur d'instances
  int position; // somme en dépôt à la banque
  void deposer(int somme) {}; // opération
  void retirer(int somme) {}; // opération
}
```

# Encapsulation

Dans la définition précédente, on a intérêt à interdire l'accès direct à la position du compte (sinon il serait facile de s'attribuer de l'argent). Ce que l'on désire donc c'est restreindre les accès aux différents membres qui composent une classe.

Il existe quatre catégories de protection :

- `private` : le membre est invisible pour quiconque (sauf pour l'instance elle-même),
- `protected` : le membre est invisible (sauf pour les instances d'objets d'une classe définie dans le même `package`, ou pour les instances d'une sous-classe),
- par défaut : le membre est invisible (sauf pour les instances d'objets d'une classe définie dans le même `package`),
- `public` : le membre est totalement visible

De plus, les méthodes peuvent être qualifiées en utilisant :

- `abstract` : ce qui conduit à déclarer sans définir la méthode (par extension la classe ne peut être qu'abstraite),
- `final` : ce qui interdit à toute sous-classe de redéfinir cette méthode,

Il existe d'autre qualificatifs, pour les méthodes ou les champs, mais il ne sont pas, à ce stade, nécessaires.

On doit garder à l'esprit que tout ce qui relève de l'implantation doit être a priori invisible !

Par exemple :

```
abstract Class Complexe {
  abstract public float module();
}
class ComplexePolaire extends Complexe  {
  private float module,angle;
  public float module() { return module; }
}
class ComplexeCartesien extends Complexe {
  private float x, y;
  public float module() { return Math.sqrt(x*x+y*y); }
}
```

Les champs sont privés afin d'interdire à quiconque de les modifier sans passer par ce qui est prévu, ou de les consulter alors qu'une autre implantation ne les contiendrait pas.

# La création d'objets

Un opérateur spécial du langage est destiné à créer des objets : `new`.

Par exemple :

```
class Vehicule {}
...
Vehicule unVehicule;
unVehicule = new Vehicule();
```

Cet opérateur construit un objet conforme à la définition de la classe, et renvoie son identité.

On peut utiliser le polymorphisme :

```
abstract class Vehicule {}
class Voiture extends Vehicule {}


Vehicule v;
```

```
v = new Voiture();
```

# Constructeurs

Il est utile de pouvoir initialiser certaines caractéristiques d'un objet au moment de sa construction (garantie de cohérence).

C'est le rôle de méthodes très particulières de le permettre : les constructeurs.

Un constructeur est une méthode ne renvoyant aucune valeur (et donc sans type, même pas `void`) dont le nom est celui de la classe.

On peut surcharger les constructeurs (en définir plusieurs) : ce qui correspond à différentes façons de construire les objets.

Par exemple :

```
class CompteEnBanque {
  private int position;
  public CompteEnBanque() {
    position = 0;
  }
  public CompteEnBanque(int depotInitial) {
    position = depotInitial;
  }
}
...
CompteEnBanque unCompte, unAutreCompte;

unCompte = new CompteEnBanque();
unAutreCompte = new CompteEnBanque(100);
```

Ou encore :

```
class CompteEnBanque {
  private int position;
  public CompteEnBanque() {
    this(0);
  }
  public CompteEnBanque(int depotInitial) {
    position = depotInitial;
  }
}
...
CompteEnBanque unCompte, unAutreCompte;

unCompte = new CompteEnBanque();
unAutreCompte = new CompteEnBanque(100);
```

# this

Dans le dernier cas, on utilise le mot-clé `this`.

C'est un concept important des objets.

Ce mot-clé fait référence à l'objet lui-même : un objet est capable de se nommer lui-même (le "je" dans un discours ego-centré).

En voici une autre utilisation :

```
class A {
  private int a;
  public void add(int a) { this.a += a; }
}
```

Un objet a parfois besoin de parler de lui, non pas comme instance de sa classe, mais de sa super-classe. C'est le mot-clé `super`.

Par exemple :

```
class A {
  public A(int v) {}
}
class B extends A {
  public B() { super(10); }
  public B(int v) { super(v); }
}
```

# Destruction des objets

Comme il a déjà été indiqué, les objets sont construits explicitement (`new`) mais détruits implicitement. Il n'y a pas d'opérateur de destruction.

Les objets sont référencés à l'aide de variables, et le système Java possède un mécanisme de suivi des références. Ainsi lorsque qu'un objet ne possède plus de références (aucun moyen de le nommer), le système peut le considérer comme destructible. Et lorsque le système le jugera nécessaire, il procédera à sa destruction.

Ce mécanisme est connu sous le nom de ramasse-miettes (garbage collector).

Toutefois, il peut être nécessaire d'effectuer certaines opérations avant de faire disparaître l'objet (libérer des ressources qu'il détiendrait encore). C'est pourquoi il existe une méthode particulière : `finalize` (héritée de `Object`).

Par exemple :

```
class A {
  private String msg;
  public A(String m) { msg = m; }
  public void finalize() {
    System.out.println("Je meurs ("+msg+")");
  }
}
...
A a = new A("un");
A b = new A("deux");
a = b; // "un" n'est plus référencé
/* Le système est libre de libérer "un" à partir d'ici */
a = null; // Encore une référence
b = null; // Dernière référence sur "deux" perdue...
```

# Liaison dynamique

C'est le mécanisme qui permet de déterminer la méthode qui sera réellement appelée à un instant donné.

Nous avons rapidement indiqué que des méthodes peuvent être redéfinies lors d'une spécialisation. Ce qui pose le problème suivant :

Étant donné un programme écrit en termes d'une classe et qui appelle une méthode, quelle sera la méthode effectivement appelée si le programme est exécuté avec des instances d'une sous-classe dans laquelle la méthode en question est redéfinie ?

La règle à retenir est que les méthodes sont attachées aux objets, et que les types ne sont que des informations destinées à vérifier des propriétés à la compilation. Donc que c'est toujours la méthode attachée à l'objet qui sera appelée.

Par exemple :

```
class A {
  protected String msg;
  public A(String m) { msg = m; }
  public void f() {
    System.out.println("Je suis "+msg);
  }
}
class B extends A {
  public B(String m) { super(m); }
  public void f() {
    System.out.println("En réalité, je suis "+msg);
  }
}
```

```
public class ex {
  public static void appelF(A a) {
    a.f();
  }
  public static void main(String aa[]) {
    A a = new A("toto");
    B b = new B("titi");
    appelF(a);
    appelF(b);
  }
}
```

# Les exceptions

Java possède (il n'est pas le seul) un mécanisme de gestion d'erreurs assez élégant : les exceptions.

Une exception est levée (`throw`) lorsque qu'une condition, qui conduirait à ne pas exécuter le code normalement, est vérifiée.

La machine Java stoppe l'exécution courante, et remonte dans la pile des appels jusqu'à un point de récupération (s'il n'y en a pas la machine s'arrête).

Il existe une construction du langage qui permet de capturer des exceptions : `try`/`catch`/`finally`.

Les exceptions sont des objets Java et donc typées. On peut donc différencier les conditions anormales.

Par exemple :

```
class Pile {
  // Qui est le crétin qui m'envoie une pile Vide ?
  public Element pop() throws Vide {
    if (isEmpty()) throw new Vide();
  }
}
class A {
  // Si la pile est Vide ce n'est pas ma faute
  public void f(Pile p) throws Vide {
    Element e = p.pop();
  }
}
```

```
class B {
  public void g() {
    Pile p = new Pile();
    A a = new A();
    try {
      a.f(p);
    } catch (Vide v) {
      System.err.println("Oups! C'est moi...");
    } catch (Exception e) {
      System.err.println("Oups! C'est quoi ca ?");
    } finally {
      System.exit(0);
    }
  }
}
```

Toute méthode levant potentiellement une exception doit le déclarer impérativement en utilisant la clause `throws`.

Tout appel à une méthode possédant une clause `throws` doit être fait :

- soit à l'intérieur d'un bloc `try` associé à un bloc `catch` avec un type compatible,
- soit à l'intérieur d'une méthode déclarant elle aussi un `throws` correspondant.

Le bloc `finally` est systématiquement exécuté lorsque l'on sort du bloc `try` : soit par une exception, soit par une condition normale.

En Java, les exceptions déclarées ne peuvent être ignorées !

Si cela semble trop contraignant, c'est pour le bien du logiciel : déclarer une erreur potentielle lors d'un calcul est un chose importante.

# Les packages standards

Les packages jouent le rôle de bibliothèques de classes, types...

On peut citer :

- `java.applet` pour créer des applets (applications embarquées)
- `java.awt` pour l'interface graphique
- `java.io` pour les entrées/sorties
- `java.lang` pour les classes de base du langage
- `java.net` pour la gestion du réseau
- `java.rmi` pour les objets distribués
- `java.security` pour la sécurité
- `java.sql` pour les bases de données
- `java.util` pour rendre la vie plus facile
- `javax.swing` pour une interface graphique plus "belle"
- `org.omg.CORBA` pour les objets distribués

Ces différents packages contiennent plusieurs dizaines de définition d'interfaces, classes, exceptions.

Il est donc recommandé de lire soigneusement :
– un tutorial,
– la documentation de référence,
– un livre,
– consulter le Web,
– pratiquer,
– ...
avant de se lancer dans un gros développement.

Une règle de vie du programmeur : dites-vous que vous n'êtes pas le premier !

Vous pouvez aussi consulter le site "`http://developer.java.sun.com/`". Vous y trouverez : des conseils, de bonnes idées, les rapports de bogues, etc.

# java.lang

Il contient les classes de base nécessaires au bon fonctionnement de la machine Java. On y trouve (entre de nombreuses autres) les définitions de :

| | | |
|---|---|---|
| Cloneable | Comparable | Runnable |
| Boolean | Byte | Character |
| Class | ClassLoader | Compiler |
| Double | Float | Integer |
| Long | Math | Number |
| Object | Package | Process |
| Runtime | SecurityManager | Short |
| String | StringBuffer | System |
| Thread | Throwable | Void |

# `java.lang.Integer`

C'est une classe d'encapsulation pour les entiers (qui ne sont pas des objets mais normalement des valeurs).

On y trouve la définition de constante : `Integer.MIN_VALUE` et `Integer.MAX_VALUE` qui donnent les valeurs extrêmes exprimables dans le type primitif associé.

Il existe deux constructeurs : `Integer(int)` et `Integer(String) throws NumberFormatException`.

Elle contient de nombreuses méthodes statiques : de conversion depuis et vers différents formats (ex : `static public String toOctalString(int i)`).

Elle contient de nombreuses méthodes d'instance : de conversion, de comparaison, etc...

Par exemple :

```
int i;
i = Integer.parseInt("56");
```

Attention les variables de type `Integer` référencent des objets :

```
int i, j;
i = 3; j = 3;
// C'est bon
if (i==j) System.out.println("C'est bon");
Integer io, jo;
io = new Integer(3);
jo = new Integer(3);
// C'est pas bon
if (io==jo) System.out.println("C'est bon");
```

Sur les variables de type primitif on teste les valeurs.

Sur les variables de type référence en teste l'identité des objets.

# java.lang.Object

- public Object()
- protected Object clone()
- public boolean equals(Object o)
- protected void finalize()
- public Class getClass()
- public int hashCode()
- public void notify()
- public void notifyAll()
- public String toString()
- public void wait()
- public void wait(long timeout)
- public void wait(long timeout, int nanos)

Il n'existe qu'une seule façon de construire un `Object`

La méthode `clone` permet d'obtenir un clone de l'objet (sous la condition qu'il déclare implémenter l'interface `Cloneable`).

La méthode `equals` permet d'implémenter l'équivalence d'objets. Cette méthode doit fournir une relation réflexive, symétrique, transitive et consistante. Aucun objet ne doit être équivalent à `null`.

La méthode `hashCode` doit être compatible avec `equals` : deux objets équivalents doivent avoir le même code de hachage.

La méthode `toString()` convertit un objet en une chaîne de caractère le représentant.

Comme `Object` est la super-classe ultime du système de type de Java, tout objet Java quelqu'il soit est conforme à cette spécification et on peut donc appeler à coup sûr (vraiment ?) n'importe laquelle de ces méthodes.

# java.lang.String

Les chaînes de caractères sont des objets dont la classe est `java.lang.String`. Une particularité de ces objets est qu'ils sont constants (non mutables).

De plus, le système Java autorise l'utilisation de l'opérateur `+` sur les `String` : c'est d'ailleurs le seul opérateur surchargé du langage.

Pour des raisons pratiques on peut utiliser le caractère `"` pour construire une `String`.

Note : les caractères qui forment une `String` sont codés sur 16 bits non signés en représentation Unicode.

Par exemple :

```
String s = "une chaîne";
System.out.println("La chaîne s vaut "+s);
```

Parmi les méthodes de la classe on trouvera :

- `public char charAt(int index)`
- `public int compareTo(String s)`
- `public String concat(String s)`
- `public int indexOf(int c)`
- `public int length()`
- `public String substring(int begin,int end)`
- `public String toLowerCase()`
- `static public String valueOf(float)`

# Les “threads”

Le système Java comprend un mécanisme d’exécution concurrente : les `Thread`.

Un `Thread` n’est qu’un objet représentant un fil d’exécution (pointeur sur instruction courante) : un `Thread` n’est donc que la représentation d’une exécution.

Le système Java possède divers mécanismes de gestion et de contrôle de ces `Thread` : un ordonnanceur avec priorité, des primitives de synchronisation, etc.

La class `java.lang.Thread` représente ces objets. De plus il existe une interface `java.lang.Runnable` qui représente le type des objets fournissant un code à exécuter à la machine Java.

# `java.lang.Runnable`

Cette interface ne contient la déclaration que d'une seule méthode : `public void run()`.

Le type représenté est celui d'objets "exécutables".

Cette méthode sera appelée lorsqu'un objet `Thread` (incarnant un objet de ce type) démarrera : elle joue donc le rôle du paramètre `void *(*start_routine, void*)` des thread POSIX.

La sortie de cette méthode conditionne donc la "vie" d'un `Thread`.

Un exemple :

```
public class ListeEntier implements Runnable {
  private int min, max;
  public ListeEntier(int min,int max) {
    this.min = min;
    this.max = max;
  }
  public void run() {
    for (int i=min; i<max; i++) {
      System.out.println("Je suis "+
        Thread.currentThread().getName()+
        " ("+i+")");
    }
  }
}
```

# java.lang.Thread

Cette classe concrête représente les objets exécutant du code.

Les méthodes de cette classe sont (entre autres) :

- `public Thread()`
- `public Thread(Runnable target)`
- `public Thread(Runnable target,String name)`
- `public String getName()`
- `public void interrupt()`
- `public static isInterrupted()`
- `public void join()`
- `public void run()`
- `public static void sleep(long millis)`
- `public void start()`
- `public void run()`
- `public static void yield()`

Par exemple :

```
public class ExListe {
  public static void main(String []a) {
  Thread t1, t2;
  ListeEntier l1, l2;

  l1 = new ListeEntier(0,1000);
  l2 = new ListeEntier(0,1000);
  t1 = new Thread(l1,"l1");
  t2 = new Thread(l2,"l2");
  t1.start();
  t2.start();
  }
}
```

Le résultat est un mélange de messages.

# La concurrence

L'utilisation de `Thread` peut conduire à des situations délicates : l'accès concurrent.

Or, il est parfois nécessaire d'interdire ou de contrôler les accès concurrents.

Le langage Java offre la qualification `synchronized` pour les méthodes : ce qui signifie qu'un `Thread` appelant une telle méthode tentera d'abord d'acquérir un "verrou" (l'attente est bloquante). Le verrou étant "relâché"à la sortie.

Il s'agit d'exclusion mutuelle : Une méthode qualifiée `synchronized` permet de garantir un accès exclusif à son code.

Pour obtenir explicitement un "verrou" sur un objet, le langage Java possède l'expression `synchronized(`*`objet`*`)` *`bloc`* .

La classe `Object` possède les méthodes `notify()` et `wait()` qui

permettent de réveiller des `Thread` en attente.

Il s'agit de synchronisation conditionnelle : Les opérations sur un objet sont bloquantes si une condition de réalisation n'est pas vérifiée.

Par exemple : je stocke si le hangar n'est pas plein, je déstocke lorsque le hangar n'est pas vide.

Soit :

```
class Stock {
  private Hangar leStock;
  public synchronized Produit destocke() {
    while (leStock.estVide()) wait();
    Produit p = leStock.retireUnProduit();
    notifyAll();
    return p;
  }
```

```
  public synchronized void stocke(Produit unProduit) {
    while (leStock.estPlein()) wait();
    leStock.rangeUnProduit(unProduit);
    notifyAll();
  }
}
```

# java.util

Il contient de nombreuses classes “utilitaires”. On y trouve (entre autres) les définitions de :

| | | |
|---|---|---|
| `Enumeration` | `Calendar` | `Date` |
| `Hashtable` | `Random` | `Stack` |
| `StringTokenizer` | `Vector` | |

# java.util.Calendar

```
import java.util.*;
public class CalTest {
  public static String myConvert(Calendar c) {
    return ""+c.get(Calendar.DAY_OF_MONTH)+"/"+
      c.get(Calendar.MONTH)+"/"+
      c.get(Calendar.YEAR);
  }
  public static void main(String [] a) {
    Calendar c = Calendar.getInstance();
    System.out.println("Aujourd'hui : "+myConvert(c));
    c.add(Calendar.DAY_OF_MONTH,37);
    System.out.println("Dans 37 jours : "+myConvert(c));
  }
}
```

# java.util.Hashtable

Cette classe implémente une structure efficace de recherche d'éléments repérés par leur clé.

Son efficacité repose sur les méthodes `hashCode` et `equals` des objets utilisés comme clés.

Par exemple :

```
import java.util.*;
Hashtable cu = new Hashtable();
cu.put("pi",new Double(Math.PI));
cu.put("e",new Double(Math.E));
cu.put("or",new Double((1.0+Math.sqrt(5))/2));
System.out.println("Le nombre d'or est : "+cu.get("or"));
```

Ou encore :

```
Hashtable mr = new Hashtable();
mr.put("google","http://www.google.com/");
mr.put("altavista","http://www.altavista.com/");
mr.put("yahoo","http://www.yahoo.fr");
System.out.println("AltaVista="+mr.get("altavista"));
```

Bref : une table de hachage se comporte comme un dictionnaire (il existe d'ailleurs à ce propos une classe abstraite `java.util.Dictionnary`.)

# Les itérateurs

Lorsqu'un objet représente une structure composée d'autres objets il est souvent nécessaire de fournir un moyen de "parcourir" la liste des composants : c'est ce que l'on appelle un itérateur.

Un itérateur est un objet qui représente un parcours de structure.

On en retrouve de différentes sortes dans le package `java.util`, mais le principal est `Enumeration`.

Il s'agit d'un type interface qui contient la déclaration de deux méthodes :

- `public boolean hasMoreElements()`
- `public Object nextElement()`

Son utilisation est la suivante :

```
import java.util.*;
public class CalTest {
  public static void listeTout(Enumeration e) {
    while (e.hasMoreElements()) {
      System.out.println("Il y a : "+e.nextElement());
    }
  }
  public static void main(String [] a) {
    Hashtable mr = new Hashtable();
    mr.put("google","http://www.google.com/");
    mr.put("altavista","http://www.altavista.com/");
    mr.put("yahoo","http://www.yahoo.fr");
    listeTout(mr.elements());
  }
}
```

Le “contrat” de la méthode `java.util.Hashtable.element()` est de fournir un itérateur parcourant la liste des valeurs associées aux clés.

On peut ainsi écrire un algorithme “générique” qui étant donné un itérateur parcours la structure qui lui est associée.

Reste une question : l'itération s'effectue-t'elle sur la structure à l'instant de l'obtention de l'itérateur ou sur la structure vivante ?

Il n'y a pas de réponse imposée : c'est un choix à faire.

# java.util.Vector

Les tableaux Java sont de taille fixée à la construction.

C'est pourquoi Java fournit les `Vector` : tableaux de taille ajustée dynamiquement. Si la sémantique est bien celle-ci, la syntaxe de Java ne permet pas d'utiliser la notation indicée `[]`. De plus, Java (qui n'est pas générique) ne peut garantir la cohérence des types.

- `public void add(int index,Object o)`
- `public void addElement(Object o)`
- `public void clear()`
- `public boolean contains(Object o)`
- `public Object elementAt(int index)`
- `public Enumeration elements()`
- `public boolean isEmpty()`
- `public Object remove(int index)`
- `public int size()`

# java.io

Le package `java.io` fournit une collection de classes permettant de représenter les objets du système de fichier hôte et des objets destinés à réaliser des entrées/sorties.

On y trouvera de très nombreuses classes. Outre la classe `File` que nous décrirons plus loin, les classes sont regroupées en deux catégories :

- les `Stream` : objets d'entrées/sorties de bas niveau sur lesquels on ne lit ou écrit que des octets.
- les `Reader`/`Writer` : objets d'entrées/sorties de haut niveau sur lesquels la lecture de base s'effectue sur des caractères : ils s'occupent donc des problèmes de (dé)codage de caractères.

On notera aussi que la plupart de ces objets réalisent des lectures/écritures en série. Les accès directs nécessitent l'utilisation d'une classe spécifique : `RandomAccessFile`.

# java.io.File

Les objets du système de fichiers hôte répertoires, fichiers) sont représentés par des instances de la classe `File`.

Parmi les méthodes on trouve :

- `public boolean canRead()`
- `public boolean createNewFile()`
- `public boolean delete()`
- `public boolean exists()`
- `public boolean isDirectory()`
- `public boolean isFile()`
- `public long lastModified()`
- `public long length()`
- `public File [] listFiles()`
- `public boolean mkdir()`
- `public boolean renameTo(File new)`

```
import java.io.*;
import java.util.*;
class PourComparer implements Comparator {
  public int compare(Object o1,Object o2) {
    String s1 = ((File)o1).getName();
    return s1.compareToIgnoreCase(((File)o2).getName());
  }
}
public class Ls {
  static public void printRef(File f) {
    if (f.isFile()) {
      System.out.println("Fichier    : "+f.getName());
    } else {
      System.out.println("Répertoire : "+f.getName());
    }
  }
```

```
static public void ls(String name) {
  File f = new File(name);
  if (f.isFile()) { printRef(f); }
  else {
    System.out.println();
    System.out.print("Répertoire : ");
    try {
      System.out.println(f.getCanonicalPath());
    } catch (IOException e) {
      System.out.println(f.getAbsolutePath());
    }
    File []files = f.listFiles();
    Arrays.sort(files,new PourComparer());
    for (int i=0; i<files.length; i++) printRef(files[i]);
  }
}
```

```
  static public void main(String []argv) {
    String []names;
    names = argv;
    if (argv.length==0) {
      names = new String[1];
      names[0] = ".";
    }
    for (int i=0; i<names.length; i++) ls(names[i]);
  }
}
```

# java.io.InputStream

Cette classe abstraite est la super classe de tous les `InputStream` et possède la déclaration/définition des méthodes suivantes :

- `public int available()`
- `public void close()`
- `public void mark(int limit)`
- `public boolean markSupported()`
- `abstract public int read()`
- `public int read(byte [] b)`
- `public int read(byte [] b,int off,int len)`
- `public void reset()`
- `public void skip(long n)`

# java.io.OutputStream

Cette classe abstraite est la super classe de tous les `OutputStream` et possède la déclaration/définition des méthodes suivantes :

- `public void close()`
- `public int flush()`
- `abstract public void write(int b)`
- `public void write(byte [] b)`
- `public void write(byte [] b,int off,int len)`

Les `Stream` définis dans `java.io` sont :

| | |
|---|---|
| `ByteArrayOutputStream` | `ByteArrayInputStream` |
| `FileOutputStream` | `FileInputStream` |
| `FilterOutputStream` | `FilterInputStream` |
| `BufferedOutputStream` | `BufferedInputStream` |
| `DataOutputStream` | `DataInputStream` |
| `ObjectOutputStream` | `ObjectInputStream` |
| `PipedOutputStream` | `PipedInputStream` |
| `PrintStream` | |
| | `LineNumberInputStream` |
| | `PushBackInputStream` |
| | `SequenceInputStream` |

# java.io.Reader

Cette classe abstraite est la super classe de tous les `Reader` et possède la déclaration/définition des méthodes suivantes :

- `public void close()`
- `public void mark(int limit)`
- `public boolean markSupported()`
- `public int read()`
- `public int read(char [] b)`
- `abstract public int read(char [] b,int off,int len)`
- `public void ready()`
- `public void reset()`
- `public void skip(long n)`

# java.io.OutputStream

Cette classe abstraite est la super classe de tous les `Writer` et possède la déclaration/définition des méthodes suivantes :

- `public void close()`
- `public int flush()`
- `public void write(char b)`
- `public void write(char [] b)`
- `abstract public void write(char [] b,int off,int len)`
- `public void write(String s)`
- `public void write(String s,int off,int len)`

Les `Reader/Writer` définis dans `java.io` sont :

| | |
|---|---|
| `BufferedWriter` | `BufferedReader` |
| `CharArrayWriter` | `CharArrayReader` |
| `FileWriter` | `FileWriter` |
| `FilterWriter` | `FilterReader` |
| `OutputStreamWriter` | `InputStreamReader` |
| `PipedWriter` | `PipedReader` |
| `ObjectOutputStream` | `ObjectInputStream` |
| `StringWriter` | `StringReader` |
| `PrintWriter` | |
| | `LineNumberReader` |
| | `PushBackReader` |

# java.net

Le package `java.net` fournit des classes destinées à implémenter des applications utilisant des objets de communications à travers le réseau.

On y trouve de nombreuses classes parmi lesquelles :

- `java.net.Socket` qui représente des objets de communication de type client,
- `java.net.ServerSocket` qui représente des objets de communication de type serveur,
- `java.net.URL` qui représente une resource désignée par un URL.

Voici donc une application sur le modèle client/serveur.

```
import java.net.*; import java.io.*;
class Service extends Thread {
  static private int incarnations;
  Socket socket;
  BufferedReader r;
  int monIncarnation;
  Service(Socket s) throws IOException {
    socket = s;
    r = new BufferedReader(new InputStreamReader(s.getInputStream()));
    monIncarnation = incarnations++;
    System.out.println("Nouveau service "+monIncarnation);
  }
  public void run() {
    boolean run = true;
    while (run) {
      try {
        String s = r.readLine();
        if (s==null)
          run = false;
        else
          System.out.println("Service "+monIncarnation+" a lu "+s);
      } catch (IOException ioe) {
        run = false;
        System.err.println("Erreur de lecture");
      }
    }
    try {
      socket.close();
    } catch (IOException ioe) {
      System.err.println("Erreur de fermeture");
    }
    System.out.println("Service "+monIncarnation+" terminé");
  }
}
```

```
public class Serveur {
  public static int PORT = 11111;
  public static void main(String argv[]) {
    try {
      ServerSocket socket;
      socket = new ServerSocket(Serveur.PORT);
      while (true) {
        new Service(socket.accept()).start();
      }
    } catch (IOException ioe) {
      System.err.println(
        "Probleme acquisition socket de service");
      System.exit(1);
    }
  }
}
```

```
import java.net.*;
import java.io.*;

public class Client {
  public static void main(String argv[]) {
    Socket s;
    PrintWriter w;

    try {
      s = new Socket("localhost",Serveur.PORT);
      w = new PrintWriter(s.getOutputStream(),true);
      for (int i=0; i<10; i++) {
        try {
          Thread.sleep(1000);
          String buf = "Voila un nombre "+i;
          System.out.println("J'envoie "+buf);
          w.println(buf);
        } catch (InterruptedException ie) {
          System.err.println("Interrupted IO");
        }
      }
      s.close();
    } catch (UnknownHostException uhe) {
      System.err.println("C'est quoi localhost ?");
      System.exit(1);
    } catch (IOException ioe) {
      System.err.println("Probleme d'acquisition sortie");
      System.exit(1);
    }
  }
}
```

# `java.awt` et `java.awt.event`

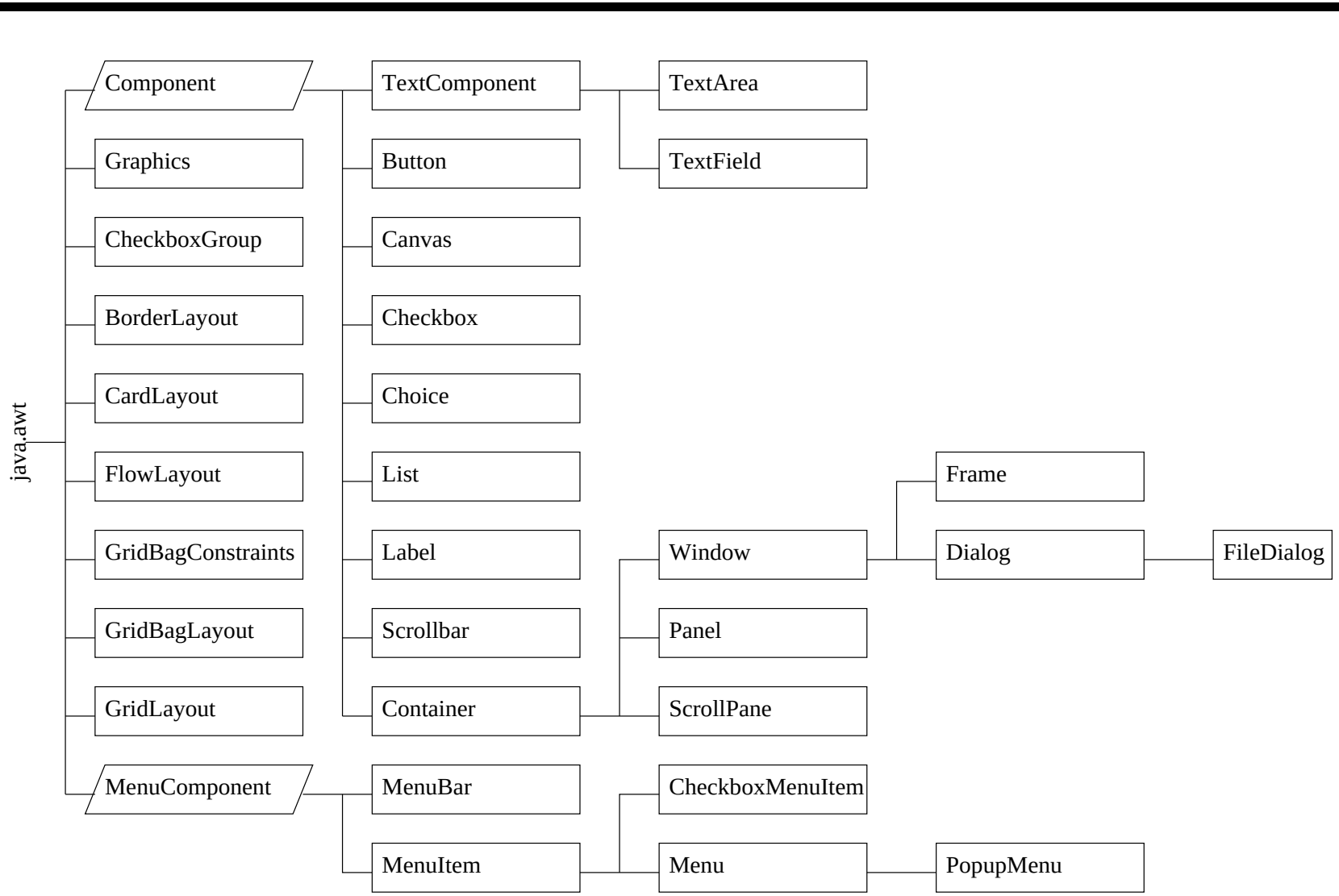
Il s'agit de 2 packages différents.

Le premier contient la définition d'objets graphiques de type interface utilisateur. AWT : Abstract Window Toolkit. Il s'agit d'objets "abstraits" car leur représentation n'est pas spécifiée (sous Windows : look Windows, sous Motif : look Motif, sous MacOS : look Mac, etc.)

Le second contient la définition d'objets représentant les évènements en provenance de l'utilisateur via l'interface graphique.

Deux classes sont à distinguer : `Component` et `Container`.

Voici un extrait de la hiérarchie des classes du package `java.awt`.


java.awt
Component
Graphics
CheckboxGroup
BorderLayout
CardLayout
FlowLayout
GridBagConstraints
GridBagLayout
GridLayout
MenuComponent
TextComponent
Button
Canvas
Checkbox
Choice
List
Label
Scrollbar
Container
MenuBar
MenuItem
TextArea
TextField
Window
Panel
ScrollPane
CheckboxMenuItem
Menu
Frame
Dialog
FileDialog
PopupMenu

# `java.awt.Component`

Un `Component` est un objet graphique pouvant être rendu (in)visible (`void setVisible(boolean)`) ou (in)actif (`void setEnabled(boolean)`), qui possède une taille (`Dimension getSize()`) et une représentation graphique (`void paint(Graphics)` et `void update(Graphics)`).

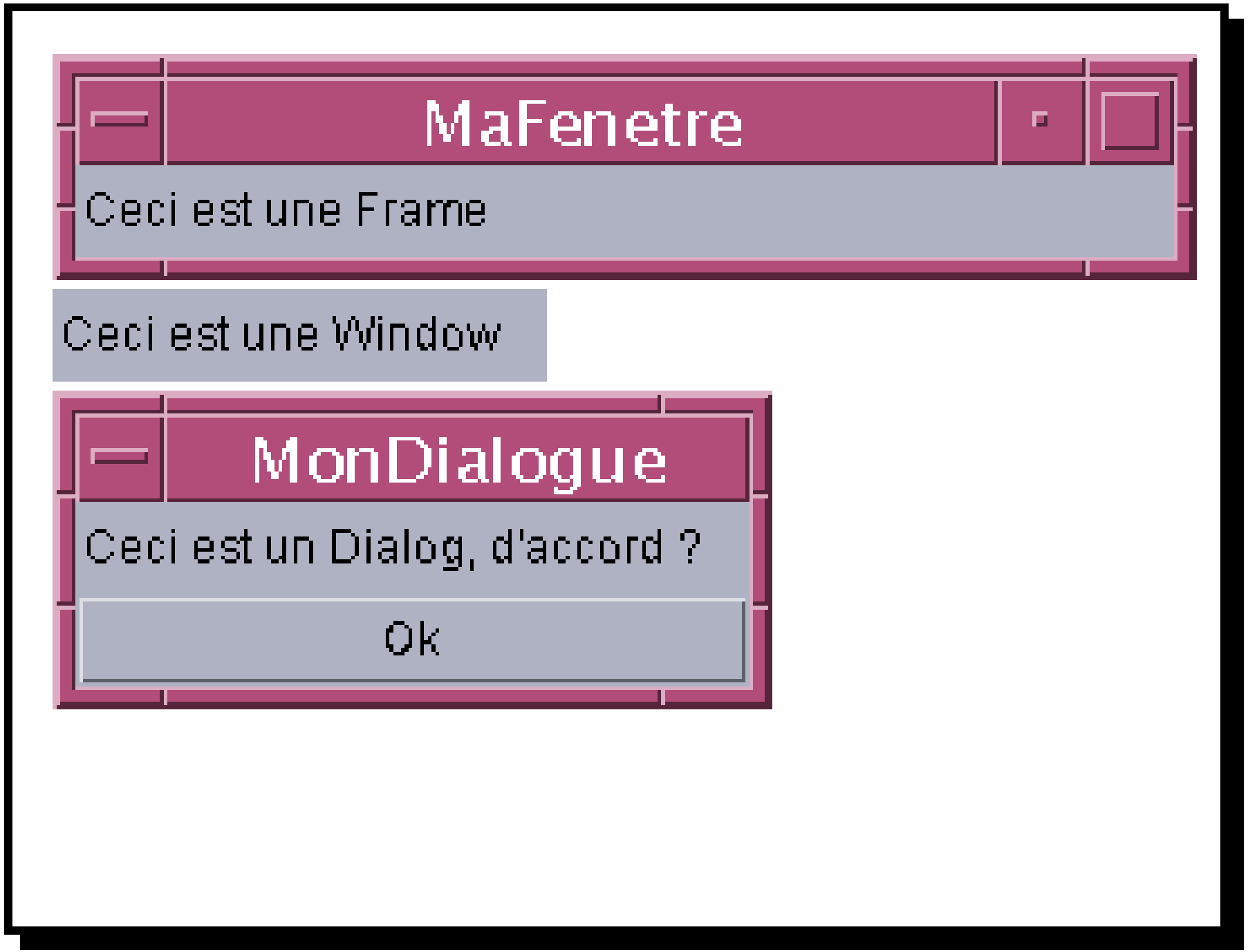
Sur un tel objet peuvent se produire différents évènements (déplacement de la souris, appui de touche clavier, etc.) Nous verrons plus loin comment “capturer” de tels évènements.

# java.awt.Container

Toute application graphique Java est construite par aggrégation de `Component` du package `java.awt`.

Certains de ces composants sont des boîtes (`Container`) dont le rôle est d'en contenir d'autres (`Component`) :

- `Window` c'est une simple fenêtre sans décorations,
- `Panel` c'est le plus simple des containers,
- `ScrollPane` c'est un container gérant lui-même des barres de défilement,
- `Frame` c'est une fenêtre principale (avec décorations),
- `Dialog` c'est une fenêtre avec décoration permettant de "dailoguer" avec l'utilisateur,
- `FileDialog` c'est une fenêtre permettant de sélectionner un fichier.


MaFenetre
Ceci est une Frame
Ceci est une Window
MonDialogue
Ceci est un Dialog, d'accord ?
Ok

Les méthodes principales de la classe `Container` sont :

- `add(Component c,...)` qui permet d'ajouter un composant dans celui-ci,
- `doLayout()` pour demander au container de ranger ses affaires,
- `getComponentAt(Point)` pour retrouver un composant contenu,
- `remove(Component)` pour sortir un composant de la boîte,
- `setLayout(LayoutManager)` pour modifier la politique de rangement

Utiliser un container est donc très simple : il suffit de le créér, de créér d'autres composants, de les ajouter et le tour est joué !

Par exemple :

```
import java.awt.*;

public class Test {
  public static void main(String a[]) {
    Frame f = new Frame("Ma fenêtre");
    Label l = new Label("Bonjour");
    Label l2 = new Label("les amis");
    f.add(l);
    f.add(l2);
    f.pack();
    f.setVisible(true);
  }
}
```

dont l'exécution produit le résultat suivant :


Où est passé le premier message ?

Le problème est que nous n'avons rien indiqué quand au placement des objets ! Le système Java a séparé la notion de placement de celle de container. Il y a différents containers et différentes façon de ranger des objets à l'intérieur.

C'est pour cela qu'il existe des classes représentant des politiques de placement.

# Le placement

Bien que le placement d'un objet puisse être calculé à volonté, pour plus de souplesse le système Java fournit des algorithmes de placement :

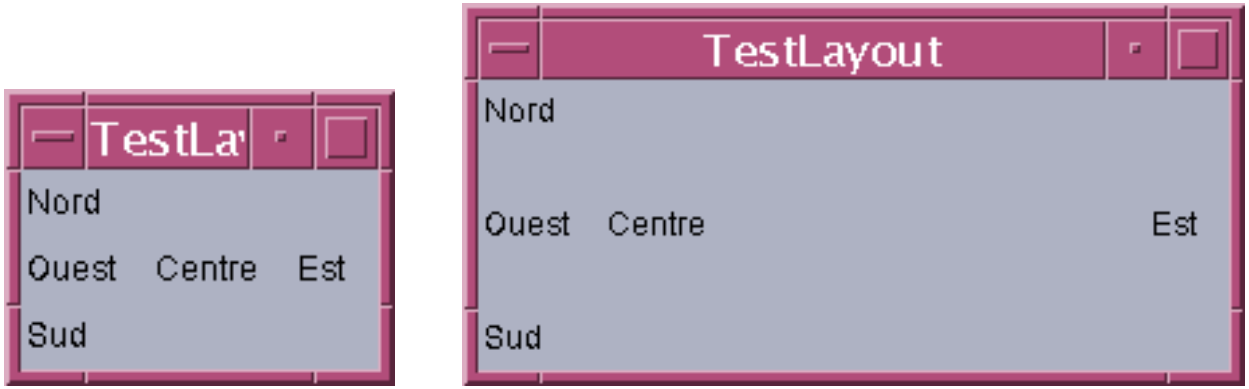
- `java.awt.BorderLayout` avec des zones prédéfinies,
- `java.awt.CardLayout` comme une pile de cartes,
- `java.awt.FlowLayout` comme du texte,
- `java.awt.GridLayout` dans une grille,
- `java.awt.GridBagLayout` sur une grille.

Lorsque le système Java demande à un container de se dessiner, celui-ci va consulter la politique de placement qui lui a été associée pour déterminer comment ranger chacun des objets.

# java.awt.BorderLayout

```
import java.awt.*;
public class TestLayout {
  public static void main(String a[]) {
    Frame f = new Frame("TestLayout");
    f.setLayout(new BorderLayout());
    Label n = new Label("Nord");
    Label s = new Label("Sud");
    Label e = new Label("Est");
    Label o = new Label("Ouest");
    Label c = new Label("Centre");
    f.add(n,BorderLayout.NORTH);
    f.add(s,BorderLayout.SOUTH);
    f.add(e,BorderLayout.EAST);
    f.add(o,BorderLayout.WEST);
    f.add(c,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
}
```

qui donne :


TestLa
Nord
Ouest
Centre
Est
Sud
TestLayout
Nord
Ouest
Centre
Est
Sud

Ce que l'on peut indiquer pour le `BorderLayout` c'est :

- que cette politique définit 5 zones,
- que les objets placés sur un bord restent collés dessus,
- que le seul composant à pâtir des redimensionements est celui du centre.

## `java.awt.CardLayout`

Cette politique permet de ranger autant d'objets que l'on veut, mais un seul d'entre eux sera visible à tout moment. Par contre des fonctionnalités sont offertes pour gérer les objets contenus comme une pile.

En général, une telle politique de placement est utilisée avec des onglets de contrôle.

# java.awt.FlowLayout

Cette politique consiste simplement à remplir l'espace comme on remplit une feuille de papier en écrivant du texte : soit de gauche à droite, puis de haut en bas.

Le code suivant :

```
import java.awt.*;
public class TestLayout {
  public static void main(String a[]) {
    Frame f = new Frame("TestLayout");
    f.setLayout(new FlowLayout());
    Label l1 = new Label("Je suis");
    Label l2 = new Label("très");
    Label l3 = new Label("content du");
    Label l4 = new Label("résultat");
    f.add(l1);
    f.add(l2);
    f.add(l3);
    f.add(l4);
    f.pack();
    f.setVisible(true);
  }
}
```

donne :

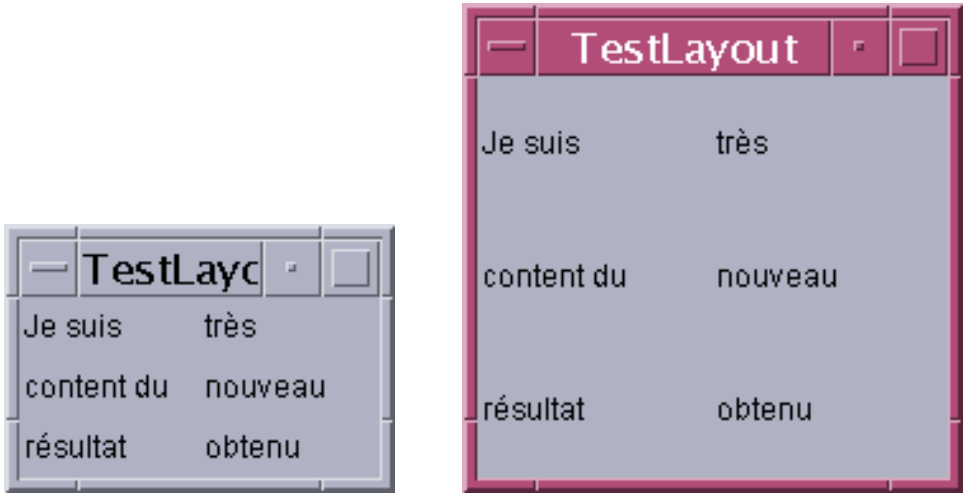

TestLayout
Je suis très content du résultat


TestLayout
Je suis très content du
résultat

# java.awt.GridLayout

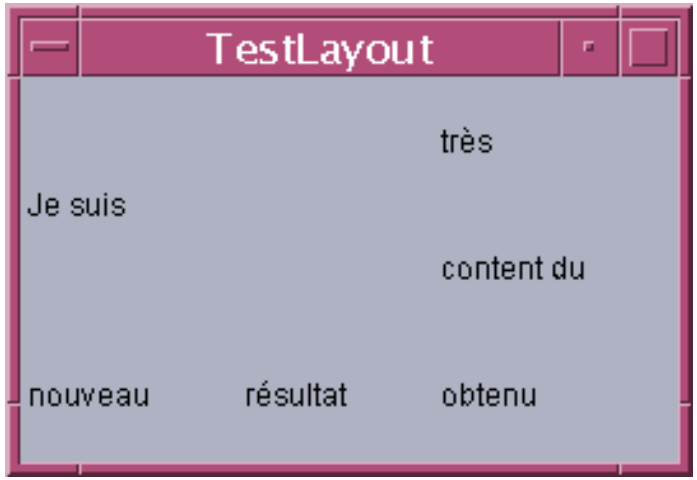
Pour cette politique, l'idée est très simple : l'espace est découpée en une grille régulière (nombre de colonnes et lignes fixés), chaque case de cette grille pouvant accueillir au plus un composant.

```
import java.awt.*;
public class TestLayout {
  public static void main(String a[]) {
    Frame f = new Frame("TestLayout");
    f.setLayout(new GridLayout(3,2));
    Label l1 = new Label("Je suis");
    Label l2 = new Label("très");
    Label l3 = new Label("content du");
    Label l4 = new Label("nouveau");
    Label l5 = new Label("résultat");
    Label l6 = new Label("obtenu");
    f.add(l1); f.add(l2);
    f.add(l3); f.add(l4);
    f.add(l5); f.add(l6);
    f.pack();
    f.setVisible(true);
  }
}
```


TestLayout
Je suis très
content du nouveau
résultat obtenu
TestLayc
Je suis très
content du nouveau
résultat obtenu

# java.awt.GridBagLayout

Cette politique utilise aussi une grille. Mais cette fois les composants se placent dessus en occupant une sous-grille choisie. Pour décrire cela il faut utiliser la classe `GridBagConstraint`.

# java.awt.Button

Un bouton est un simple objet graphique sur lequel il est posible d'agir en cliquant pour obtenir une réponse.

```
import java.awt.*;
public class Bouton {
  public static void main(String a[]) {
    Frame f = new Frame("Ma Fenetre");
    Button b = new Button("Cliquez-moi !");
    f.add(b);
    f.pack();
    f.setVisible(true);
  }
}
```

Dans ce programme, rien n'indique que nous sommes intéressé par les évènements qui pourrait être produits par l'utilisateur sur l'objet bouton.

Nous pouvons "cliquer" sur celui-ci, un effet graphique sera probablement réalisé, mais rien d'autre de particulier ne se passera.

# Les évènements

Les objets graphiques du package `java.awt` sont destinés à permettre certaines interactions de la part de l'utilisateur.

Chaque objet graphique possède un ensemble de réactions possibles (c'est justement ce qui fait ses particularités). Parmi celles-ci existe la possibilité de s'intéresser (écouter) aux actions "logiquement" (évènements) définies.

Un évènement est un objet abstrait représentant une action logique sur un composant graphique. C'est une abstraction qui ne se préoccupe pas de connaître les détails du déclenchement de l'action.

Exemple : pour un bouton, l'évènement intéressant à capturer est le fait qu'un utilisateur a appuyé dessus ; mais pas de se préoccuper s'il l'a fait par l'intermédiaire de la sourie, d'un raccourci clavier ou d'une autre manière. Ce qui est effectivement important est qu'il a appuyé dessus.

Une suite d'évènement matériels (déplacement de la souris, appui sur une touche du clavier, etc) est capturée par le système graphique Java, qui redistribue chacun d'entre eux aux composants graphiques concernés, lesquels interprétent cette suite pour en extraire des évènements logiques représentant les actions fondamentales du composant.

Ces actions fondamentales sont incarnées sous la forme d'évènements (`java.awt.event`) qui sont alors tranmis à tout objet s'étant préalablement déclaré intéressé à les recevoir.

# java.awt.event.ActionEvent

La classe `java.awt.event.ActionEvent` représente l'évènement fondamental se produisant sur un bouton : à savoir "on a appuyé dessus".

Un tel objet possède les méthodes importantes suivantes :

- `String getActionCommand()` qui représente une commande associée à l'objet graphique,
- `int getModifiers()` qui représente l'état des touches spéciales du clavier,
- `Object getSource()` qui représente le composant graphique "source" de l'évènement.

# java.awt.event.ActionListener

L'interface `java.awt.event.ActionListener` est le type des objets intéressés pour "écouter" des boutons.

Elle impose la définition d'une seule méthode :

– `public void actionPerformed(ActionEvent)` qui sera appelée lorsque l'évènement décrit par le paramètre se sera produit sur le bouton.

Pour que l'appel à `actionPerformed` se produise, il faut préalablement s'être déclaré auprès du bouton. C'est pourquoi les objets `Button` possèdent une méthode
`public void addActionListener(ActionListener)`

Un code plus complet serait :

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

class Capteur implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("J'entends!");
  }
}

public class Bouton2 {
  public static void main(String a[]) {
    Frame f = new Frame("Ma Fenetre");
    Button b = new Button("Cliquez-moi !");
    Capteur c = new Capteur();
    b.addActionListener(c);
    f.add(b);
    f.pack();
    f.setVisible(true);
  }
}
```

Note : il est possible "d'enregistrer" plusieurs `ActionListener`. Chacun d'eux sera appelé à tour de rôle (dans l'ordre d'enregistrement).

On peut se désenregistrer en utilisant

`public void removeActionListener(ActionListener)`

# Les `Listener`

Les `Listener` principaux prédéfinis sont :

- `ActionListener` pour recevoir des `ActionEvent`
- `AdjustmentListener` pour recevoir des `AdjustmentEvent`
- `ComponentListener` pour recevoir des `ComponentEvent`
- `FocusListener` pour recevoir des `FocusEvenet`
- `ItemListener` pour recevoir des `ItemEvent`
- `KeyListener` pour recevoir des `KeyListener`
- `MouseListener` pour recevoir des `MouseEvent`
- `MouseMotionListener` pour recevoir des `MouseEvent`
- `TextListener` pour recevoir des `TextEvent`
- `WindowListener` pour recevoir des `WindowEvent`

Les `Listener` sont alors enregistrés en utilisant à chaque fois une méthode de nom `add*Listener`.

# Les `Event`

Les `Event` principaux prédéfinis sont :

- `ActionEvent` représentant une simple “action” (déclenchement),
- `AdjustmentEvent` représentant un ajustement de valeur,
- `ComponentEvent` représentant des modifications du composant,
- `FocusEvenet` représentant l'activation du clavier,
- `ItemEvent` représentant la sélection d'un item dans une liste,
- `KeyListener` représentant la frappe d'une touche du clavier,
- `MouseEvent` représentant des évènements en provenance de la souris,
- `TextEvent` représentant des saisies de texte,
- `WindowEvent` représentant des évènements produits sur une fenêtre.

# La création de composants

Il existe deux techniques de création de composants graphiques : en utilisant le composant "vierge" `Canvas` ou en utilisant la classe `Component`. La différence est subtile mais pas fondamentale : un objet `Canvas` est attaché à une fenêtre de l'interface hôte alors qu'en sous-classant `Component` l'objet n'est qu'une illusion.

Un `Component` fournit deux méthodes permettant au système graphique d'en connaître l'allure :

- `public void paint(Graphics)` appelée par le système graphique,
- `public void update(Graphics)` appelée en réponse à une demande de l'utilisateur.

Pour créer son propre composant il est nécessaire de redéfinir ces méthodes. Mais attention : elles peuvent être appelées fréquemment c'est pourquoi il est nécessaire de ne pas y faire trop de calculs.

De plus, pour qu'un composant s'intègre normalement à l'interface graphique de Java, trois méthodes essentielles doivent être redéfinies :

- `public Dimension getMinimumSize()` qui permet au système Java de déterminer la place minimale occupée par ce composant,
- `public Dimension getMaximumSize()` qui permet au système de déterminer la place maximale pouvant être occupée par ce composant,
- `public Dimension getPreferredSize()` qui indique au système la place normalement occupée par ce composant.

# java.awt.Graphics

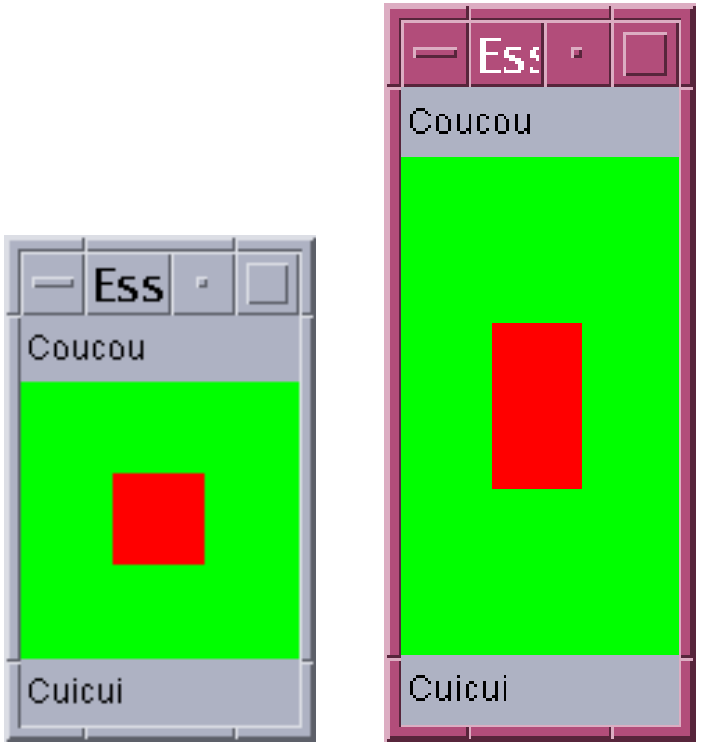
Cette classe abstraite représente ce que l'on appelle un contexte graphique : c'est à dire un objet permettant de dessiner.

On y trouve (entre autres) les méthodes suivantes :

- `public void drawRect(int,int,int,int)`
- `public void drawString(String,int,int)`
- `public void fillRect(int,int,int,int)`
- `public void setColor(Color)`
- `public void setFont(Font)`
- `public void setXORMode(Color)`

Voici un exemple de définition d'un composant :

```
import java.awt.*;
class MyComponent extends Component {
    private Dimension minDim = new Dimension(100,100);
    private Dimension maxDim = new Dimension(500,500);
    private Dimension preDim = minDim;
    public Dimension getMinimumSize() { return minDim; }
    public Dimension getMaximumSize() { return maxDim; }
    public Dimension getPreferredSize() { return preDim; }
    public void paint(Graphics g) {
        Dimension d = getSize();
        g.setColor(Color.green);
        g.fillRect(0,0,d.width,d.height);
        g.setColor(Color.red);
        g.fillRect(d.width/3,d.height/3,d.width/3,d.height/3);
    }
}
public class CompTest {
    public static void main(String []argv) {
        Frame f = new Frame("Essai");
        f.setLayout(new BorderLayout());
        MyComponent c = new MyComponent();
        Label l = new Label("Coucou");
        Label l2 = new Label("Cuicui");
        f.add(c,BorderLayout.CENTER);
        f.add(l,BorderLayout.NORTH);
        f.add(l2,BorderLayout.SOUTH);
        f.pack();
        f.setVisible(true);
    }
}
```


Ess
Coucou
Cuicui
Ess
Coucou
Cuicui