

Introduction rapide aux objets et à Java

Prolégomènes du DESS TNI de Montpellier
Septembre 2003 – Version 1.1

Jacques Ferber
LIRMM - Université de Montpellier II

ferber@lirmm.fr
<http://www.lirmm.fr/~ferber>

1

Plan

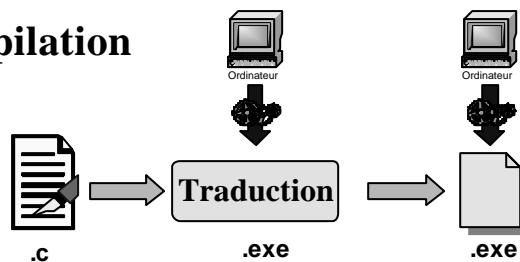
- ❖ Présentation générale de Java
- ❖ Classes
- ❖ Héritage
- ❖ Collections et Vector
- ❖ Packages
- ❖ Modifiers
- ❖ Exceptions

Le langage Java

- ❖ Langage objet apparu dans les années 1995
- ❖ Comprend:
 - Une machine virtuelle
 - Un compilateur vers la machine virtuelle
 - Des bibliothèques de classes

La notion de machine virtuelle #1

Compilation

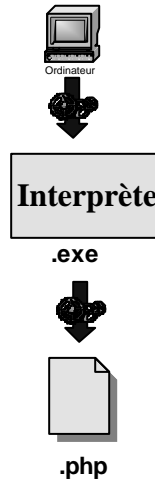


- ❖ Avantages
 - Rapidité
- ❖ Inconvénient
 - Manque de portabilité
 - Rigidité

La notion de machine virtuelle #2

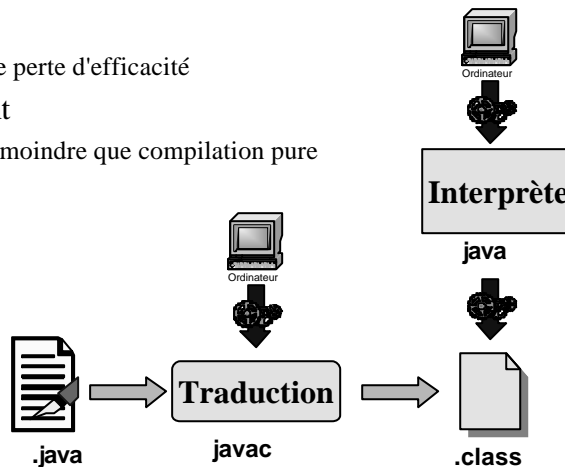
Interprétation

- ❖ Avantages
 - Souplesse
 - Portabilité
- ❖ Inconvénient
 - Lenteur



La notion de machine virtuelle #3

- ❖ Avantages **Machine virtuelle: l'exemple Java**
 - Portabilité
 - Souplesse
 - Pas trop de perte d'efficacité
- ❖ Inconvénient
 - Efficacité moindre que compilation pure



Quelques caractéristiques du langage Java

- ❖ **Syntaxe simple**
 - Syntaxe ressemblant globalement à C/C++ (sans le goto et quelques restrictions)
- ❖ **Sémantique assez classique**
 - Les tableaux sont "presque" des objets (comme en Scheme ou en Smalltalk)
 - les objets connaissent leur classe
 - Mais problèmes de typage (cast, etc... voir plus loin)

Caractéristiques générales #2

- ❖ **Robuste**
 - Pas d'accès direct à la mémoire
 - Garbage collector
 - Compilateur relativement contraignant (mécanisme de typage fort)
 - Gestion des exceptions objets
- ❖ **Sécurisé**
 - Prise en charge par la sémantique du langage et la machine virtuelle (pas possible de passer outre)
 - Vérification du byte-code, class loader, security manager (accès aux ressources)
- ❖ **Multi-thread**
 - Permet de concevoir des programmes qui donnent l'impression de s'exécuter en parallèle
 - Mécanismes de synchronisation (par moniteurs)
 - Mais quelques petites différences d'implémentation

Caractéristiques du langage #3

- ❖ Grande bibliothèque de classes
 - Java 1.4 (1.3) 2723 (1840) classes réparties en 135(76) packages représentant environ une vingtaines de domaines
 - Graphisme (2D), interfaces, entrées-sorties, collections, introspection, applets, threads, gestion de zip, jar,...
 - Bases de données (JDBC), réseau (sockets, URL), rmi, nommage de services (JNDI), CORBA (IDL)
 - son, gestion de la sécurité, internationalisation
 - XML, expressions régulières, crypto
 - + d'autres bibliothèques:
 - J2EE, JSP, Servlet,
 - Graphisme 3D
 - + toutes celles qu'on trouve sur le web

Caractéristiques #4

- ❖ La JVM de Java permet l'implémentation de nombreux langages qui ont accès aux bibliothèques de Java:
 - Python (jython)
 - Lisp/Scheme
 - Java interprété (BeanShell)
 - Langage à base de règles (Jess)
 - Prolog
 - Smalltalk
 - Forth, Logo, langages fonctionnels, langages de scripts, etc.
 - Basic, Fortran
 - Cette liste n'est pas exhaustive voir:
 - <http://www.robert-tolksdorf.de/vmlanguages.html>
- ❖ Java n'est pas seulement un langage: c'est une plate-forme!!

Syntaxe

❖ Syntaxe de base identique à celle de C/C++

```
{ <expr1>  
  <expr2>  
  <expr3> }
```

```
while (<condit>)  
  <expr1>
```

```
switch(expr){  
  case <const1> : <expr1>; break;  
  case <const1> : <expr1>; break;  
  ...  
  default : <expr>  
}
```

```
do <expr1>  
  while (<condit>)
```

```
if (<condit>)  
  <expr1>  
else  
  <expr2>
```

```
for (<init>;<tantque>;<incr>)  
  <expr1>
```

```
Comparateurs  
<, >, <=, >=, ==, !=  
Booléens  
&&, ||, !  
Arithmétiques  
+, -, /, *, %
```

```
<var> = <expr>;
```

❖ Comme en C, le ';' est un terminateur d'instruction

Types en Java

❖ Types primitifs

- Booléen: **boolean**
- Nombres entiers: **byte, short, int, long** (de 1 à 8 octets)
- Nombres flottants: **float, double** (de 4 à 8 octets)
- Caractère: **char** (2 octets Unicode)

❖ Objets

- Instance de classes (voir plus loin)
- Note: les chaînes de caractère et les tableaux sont des objets

Déclaration de variables et affectation

- ❖ On peut déclarer une variable n'importe où lors de sa première utilisation

```
int a = 34;  
int b = a + 1;  
a = a + b;
```

- ❖ Premier programme:

```
class Test {  
    public static void main(String[] args){  
        int r = 0;  
        for (int i=0;i<=10;i++){  
            r = r + i;  
        }  
        System.out.println("somme : "+r);  
    }  
}
```

L'approche objet

- ❖ La pensée objet est "naturelle"
- ❖ La pensée objet est difficile

La pensée objet est «naturelle»

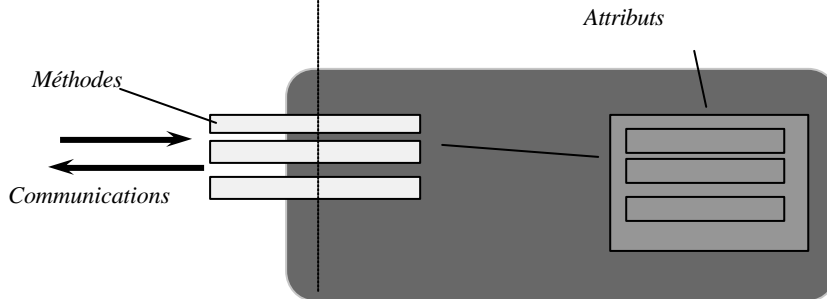
- ❖ Considère que le monde est composé d'objets
- ❖ Ces objets sont identifiables individuellement.
- ❖ Ces objets sont en relation avec d'autres objets
- ❖ Ces objets peuvent être abstraits sous la forme de «types» que l'on appelle classes
- ❖ On peut organiser ces classes sous la forme de hiérarchies

La pensée objet est difficile

- ❖ Penser en termes d'abstractions conceptuelles
 - ❖ Penser en termes d'entités relativement autonomes
 - ❖ Penser en termes de cadres (frameworks) et de composants réutilisables
- ⇒ Notre culture et notre enseignement technique et scientifique ne nous ont pas bien préparé à cette pensée

Pourquoi les objets?

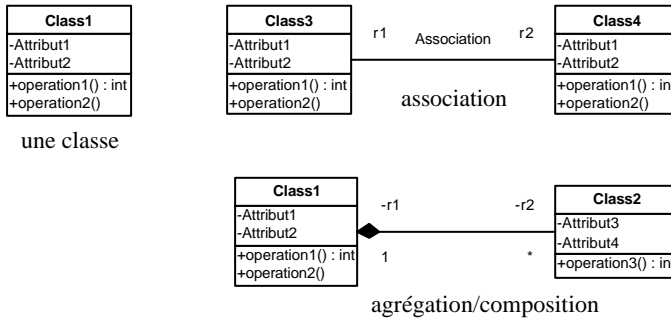
- ❖ Un objet est défini par son identité et par sa classe qui décrit:
 - Sa structure (ses attributs)
 - Son comportement (ses méthodes)
- ❖ Relation classe-instance
 - La classe est un modèle d'objet - un objet est une instance de classe
 - La classe décrit la structure et le comportement des objets
 - Les objets (instances) ne comportent que leur état (valeur des attributs)



UML

- ❖ Notation UML (Unified Modelling Language)
 - Issue des notations de OMT(Rumbaugh), OOD (Booch) et Objectory (Jacobson)
- ❖ Norme de l'OMG
- ❖ Plusieurs types de diagrammes:
 - Diagrammes de classes ———— très employé!!
 - Diagrammes de séquence
 - Diagrammes de collaboration
 - Diagrammes de cas d'utilisation
 - Diagrammes de déploiement
 - Diagrammes de composants ———— moins employés
 - Diagrammes états transitions

diagrammes de classe #1



une classe

agregation/composition

Exemple #1.1 Compte en banque

❖ L'exemple de base... ☺

```
class Compte {
    long solde;
    long numero;
    Compte(long s, long num) {
        solde = s;
        numero = num;
    }

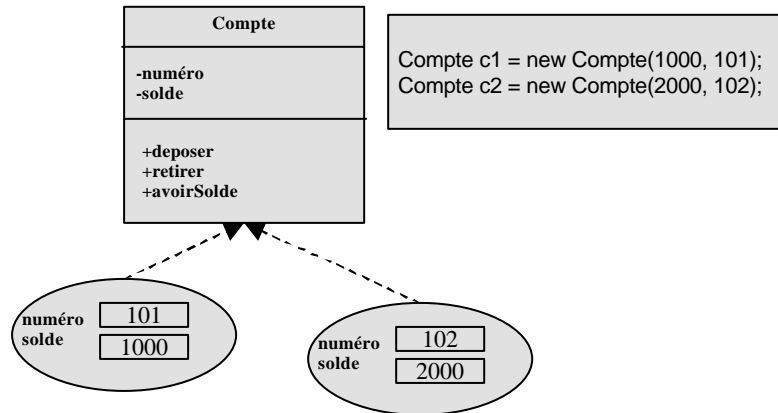
    long avoirSolde(){return(solde);}
    long avoirNumero(){return(numero);}

    void deposer(long s){ solde=solde +s;}
    void retirer(long s){solde=solde-s;}
}
```

Compte
-solde: long
-numero: long
+deposer(s:long): void
+retirer(s:long): void
+avoirSolde(): long

Classe et instance

- ❖ La classe définit un modèle d'objet
- ❖ Un objet est une "instance" (exemplaire) d'une classe



Invocation de méthode

- ❖ Invoquer une méthode (on dit aussi "envoyer un message") consiste à appliquer la méthode associée à la classe de l'objet receveur

`<expr>.<nom methode>(<arg1>,...,<argn>)`

- ❖ `<expr>` retourne un objet, le receveur du "message"
- ❖ **Note:** la notion de "message" est seulement métaphorique. Il ne s'agit que d'un appel à une "procédure" locale à la classe de l'objet receveur.

Exemple #1.2

```
Compte c1 = new Compte(1000, 101);
Compte c2 = new Compte(2000, 102);

c1.deposer(100);
c2.deposer(200);
c1.retirer(50);

System.out.println("solde de c1 : "+c1.avoirSolde()); → 1050
System.out.println("solde de c2 : "+c2.avoirSolde()); → 2200
```

Les chaînes de caractères

- ❖ Les chaînes de caractères sont des objets

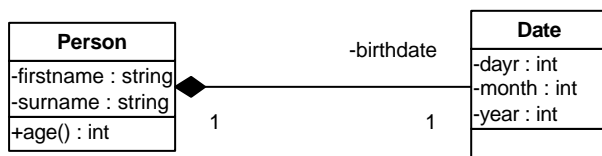
```
String s1 = "bonjour";
String s2 = "les copains";

String s3 = s1 + " " + s2;    // concaténation

Char c = s3.charAt(3);        // c = 'j'
s2 = s1.substring(3,6);      // s2 = "jou"
```

Exemple #2.1

❖ Une personne...



Exemple #2.2

```
class Date {
    int year;
    int month;
    int day;

    Date(int d, int m, int y){
        year = y;
        month = m;
        day = d;
    }
    int getYear(){return year;}
    int getMonth(){return month;}
    int getDay(){return day;}
}
```

```
class Person {
    Date birthdate;
    String firstname;
    String surname;

    int age(Date today){
        return today.getYear() - birthdate.getYear()
    }
    Person(String p, String n, Date d){
        firstname = p;
        surname = p;
        birthdate = d;
    }
    Date getDate(){return birthdate;}
}
```

```
class Test2 {
    public static void main(String[] args){
        Person p = new Person("Jean", "Dupond", new Date(6,1,1966));
        System.out.println("presentation de : "+p);
        Date d = p.getDate();
    }
}
```

Les tableaux

❖ Caractéristiques:

- Les tableaux sont des objets qui contiennent des références vers d'autres objets
 - Comme en Lisp/Scheme ou en Smalltalk
- Les tableaux ont un type tableau: <type>[]
- Tous les éléments du tableau doivent avoir le même type
- Les tableaux ont une taille fixée lors de leur création

Les tableaux: exemples

```
int[] ta = {1, 3, 5, 7, 11};
int a = ta[2]; // a = 5

String[] ts1 = {"toto", "titi", "fifi"};
String[] ts2;
ts2 = ts1; // copie de la référence uniquement
ts2[1] = "loulou";
String s = ts1[1]; // s = "loulou"

Person[] tperson = new Person[100]; // crée un tableau vide de 100 slots
tperson[0]=new Person("Jean","Dupond", new Date(6,1,1966));
...

int b = tperson[0].age(); // b = 37
```

Surcharge des opérations

- ❖ Une méthode ou constructeur est caractérisé par sa signature et pas seulement pas son nom:

<type_{retour}> nom (<type_{arg1}>...<type_{argn}>)

```
class Rectangle {
    double longueur, largeur;
    Rectangle(double lo, double la){
        longueur = lo; largeur = la;
    }
    double perimetre(){return longueur + largeur;}
    double perimetre(double r){return r*perimetre();}

    double surface(){return longueur * largeur;}
    double surface(double r){return this.surface()*r;}
}
```

This

- ❖ la pseudo-variable 'this' représente l'objet receveur d'un message et donc l'objet en cours
 - identique au 'self' de Smalltalk, et au 'this" de C++ et Simula.

```
class Truc {
    int a, b;
    Chose c;

    void m(){...}
    void p(Chose c){
        this.m();
        this.c = c;
        m();
    }

    Truc(int a){
        this.a = a;
    }
}
```

```
class Chose {
    Truc t;

    Chose(Truc t){
        this.t = t;
        t.p(this);
    }
}
```

- ❖ Note:

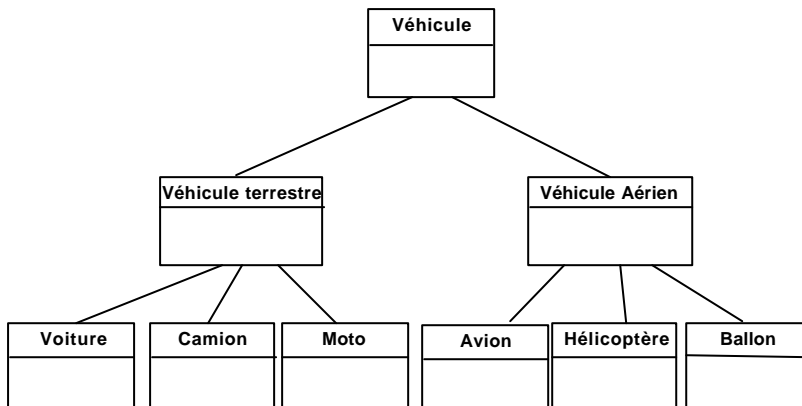
p() ≡ this.p()

a ≡ this.a()

La notion de classification et l'héritage

- ❖ Décrire un univers, c'est définir une taxonomie de classes pour catégoriser les «types de choses» de l'univers du métier et des systèmes
 - Définition d'un ordre des choses à partir d'une classification des entités par «genus et differentia» (genre et différences)
 - Idée très ancienne (3^{ème} siècle av J.C.)
- ❖ Mécanisme:
 - les attributs et les méthodes sont "recopiés" des super classes vers les sous-classes

Héritage



Définition d'héritage en Java

- ❖ Héritage simple: une classe ne peut hériter que d'une classe
- ❖ Attributs et méthodes sont "recopiés" dans les sous-classes
- ❖ Toute classe hérite d'une classe (par défaut Object)

```
class Rectangle {
    double longueur, largeur;
    Rectangle(double lo, double la){
        longueur = lo; largeur = la;
    }
    double perimetre(){return longueur + largeur;}
    double surface(){return longueur * largeur;}
}

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

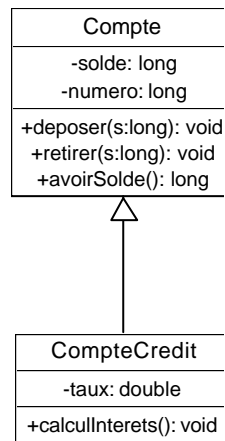
Carre c = new Carre(5);
double s = c.surface();    // 25
```

Redéfinition de méthode et 'super'

```
class CompteCredit extends Compte {
    double taux=0.05;
    CompteCredit(long s, long num, double t){
        super(s,num);
        taux = t;
    }

    void calculInterets(){
        deposer(Math.round(avoirSolde()*taux));
    }

    void retirer(long s){
        super.retirer(s);
        super.retirer(1);
    }
}
```



La classe Object

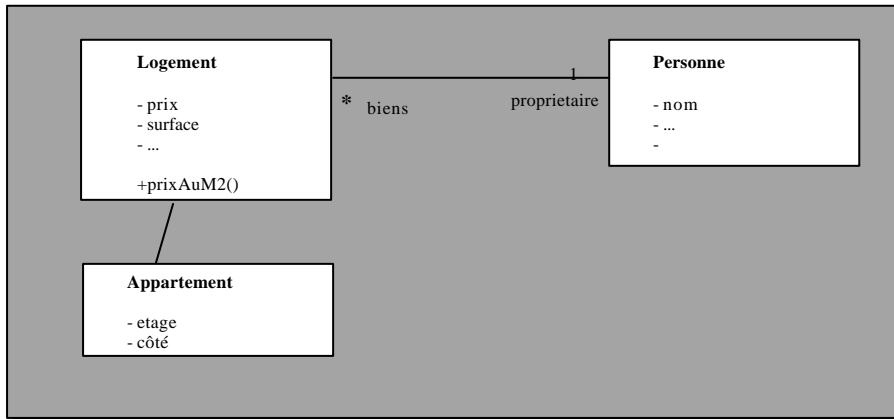
- ❖ Racine de l'héritage: toutes les classes (même les chaînes de caractères et les tableaux) en dérivent

- ❖ Méthodes importantes
 - boolean equals(Object o)
 - String toString()

Transformation des diagrammes en classes

- ❖ Les attributs UML restent des attributs (on parle aussi de champs dans ce cas)
- ❖ Les associations (relations) sont transformées en attributs (champs) contenant
 - Des références vers d'autres objets (cas des associations mono- valuées)
 - Des collections vers d'autres objets (cas des associations multi- valuées).
- ❖ Les opérations sont décrites sous la forme de méthodes.

Exemple de transformation UML - classes



Transformation UML -> Classes

```
class Logement {
    int prix;
    int surface;
    ...
    Personne proprietaire;

    int prixAuM2() {
        ...
    }
}
```

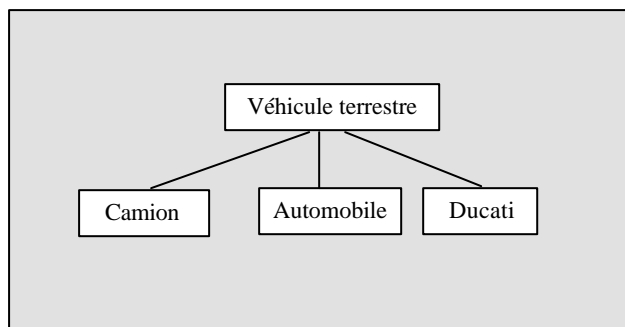
```
class Appartement extends
Logement {
    int etage;
    String position;
}
```

```
public class Personne {
    String nom;
    String prénom;
    ...
    Logement biens[ ];
}
```

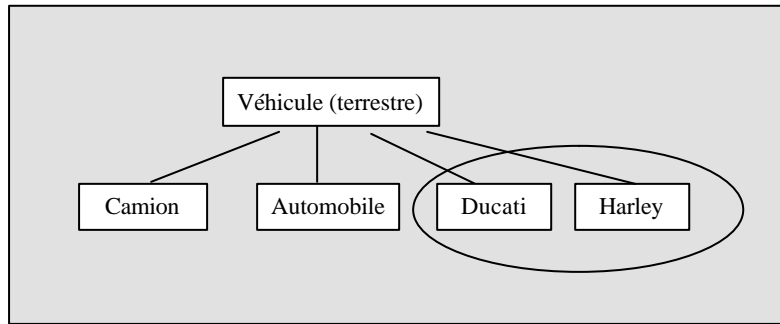
Équilibrer les classifications

- ❖ On dit qu'une classification est équilibrée si, à tous les niveaux, toutes les sous-classes d'une classe sont (à peu près) de même niveau conceptuel

Exemple de classification mal équilibrée



Conséquences



Classes abstraites

- ❖ Une classe abstraite est une classe qui ne peut pas avoir d'instance
 - Comporte des méthodes abstraites éventuellement
 - En Java une méthode abstraite est une méthode qui n'a pas de code. A distinguer des méthodes vides qui ne font rien...

```
abstract void p(); // est abstraite
void p(){} // méthode vide, ne fait rien..
```
 - Une classe abstraite peut contenir des attributs et des méthodes
 - Exemple: la classe Component en Java

Classes abstraites et terminales

Abstraction

- ❖ Classe abstraite: qui ne peut pas avoir d'instance.
- ❖ Classe concrète: qui peut avoir des instances

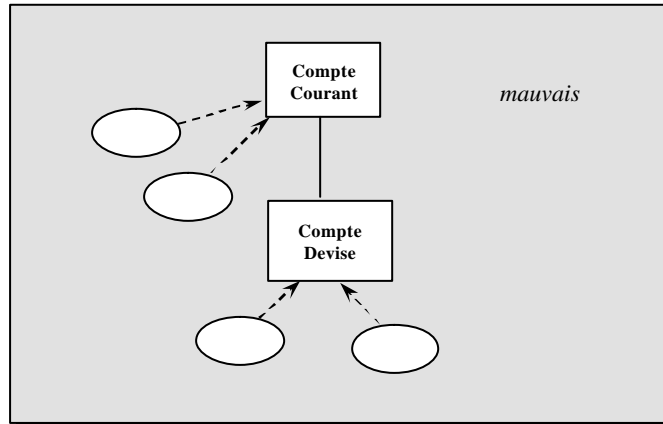
Positionnement

- ❖ Classe terminale: qui ne possède pas de sous classes
- ❖ Classe non-terminale: qui possède des sous-classes

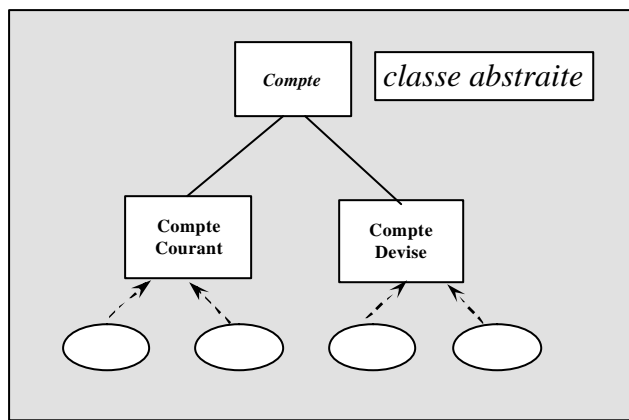
Règle d'utilisation des classes abstraites

- ❖ Une règle:
 - faire en sorte que seules les classes terminales soient concrètes, et que les classes non-terminales soient abstraites.
- ❖ Permet d'avoir des classifications plus équilibrées (et donc plus stables et extensibles)

Mauvaise classification



Bonne classification



Les classes abstraites

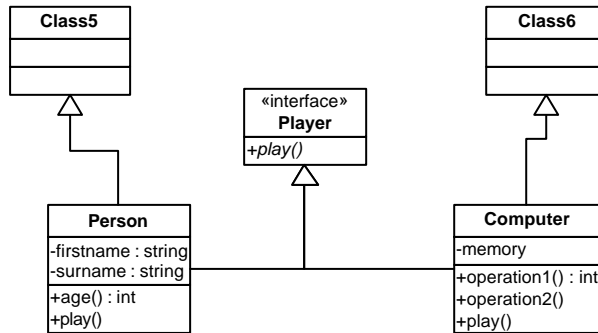
- ❖ **Les classes abstraites constituent le coeur des applications**
- ❖ **Ne pas hésiter à faire des classes abstraites**
- ❖ *C'est là que l'essentiel se passe*

Interfaces

- ❖ **Interface**
 - aspect (rôle?) qu'une classe peut jouer
 - l'interface décrit le comportement d'une classe
 - une interface ne comprend aucun code
- ❖ **Implémentation d'une interface**
 - Une classe si elle accepte d'implémenter une interface, doit effectivement implémenter (donner du code) pour toutes les méthodes déclarées dans l'interface
 - Contrat vérifié par le compilateur

Exemple

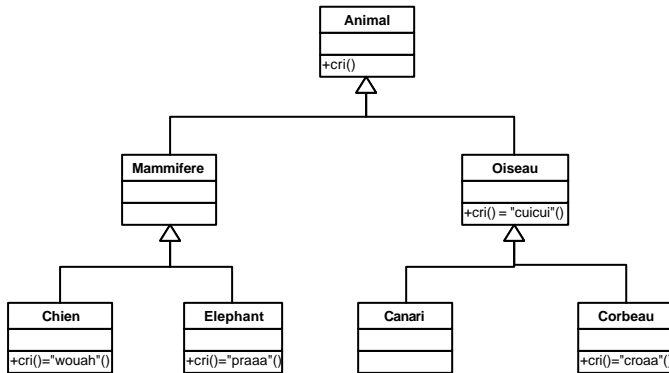
- ❖ Une personne et un ordinateur peuvent être considérés comme des joueurs
 - Ils doivent implémenter la méthode `play()`



Le typage

- ❖ Java est un langage fortement typé:
 - Toute variable a un type connu par le compilateur
 - Toute expression a un type connu par le compilateur
 - Vérification de type sont faites par le compilateur
- ❖ Un type peut être:
 - Un type primitif
 - Une classe
 - Une interface
 - Un tableau de type
- ❖ Une variable n'accepte un objet que si son type est \geq au type de l'expression:
 - `a = e; // type(a) >= type(e)`

Exemple de typage



```
Animal a = new Elephant(); // OK
Mammifere m = a;           // Erreur de compilation

Oiseau o = new Corbeau(); // OK
a = o;           // OK
a.cri()          → quel est le cri de l'animal en question?
```

Cast

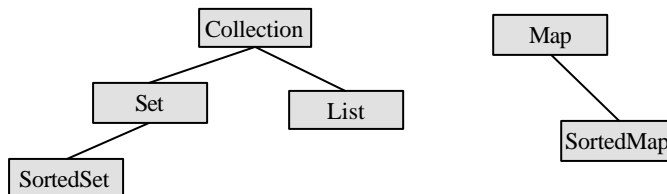
- ❖ Permet d'indiquer au compilateur que le type n'est pas celui qu'il croit
 - C'est un contrat entre le compilateur et le programmeur!!
 - `<t> e` // où `<t>` est un type.
l'expression a maintenant le type `<t>`
 - valable uniquement si `<t> = type(e)`

```
Animal a = new Elephant(); // OK
Mammifere m;
m = a; // Erreur de compilation
m = (Mammifere) a; // OK à la compilation et à l'exécution

Oiseau o;
o = (Mammifere) a; // OK à la compilation mais erreur à l'exécution
o = (Oiseau) m; // Erreur à la compilation, interdit
```

Collections et Vector

- ❖ Les collections sont des objets qui servent à "contenir" d'autres objets
- ❖ Implémentent les structures de données classiques: listes, piles, ensembles, tables, etc.
- ❖ Les collections font maintenant partie d'un "framework" collection.



La classe Vector

```
public class Vector extends AbstractList implements List {
    // opérations de base
    int size();
    boolean isEmpty();
    boolean contains(Object o); // appartenance
    boolean add(Object o);
    boolean remove(Object o);

    Object elementAt(int i); // retourne le ième élément
    Object firstElement();
    Object lastElement();

    boolean containsAll(Collection c); // inclusion
    boolean addAll(Collection c); // union
    boolean removeAll(Collection c); // différence
    void clear();

    Object[] toArray();
}
```

Exemple d'utilisation de Vector

```
Vector v = new Vector();
Compte c1 = new Compte(2000, 103);
v.add(c1);
v.add(new Compte(3000,104));

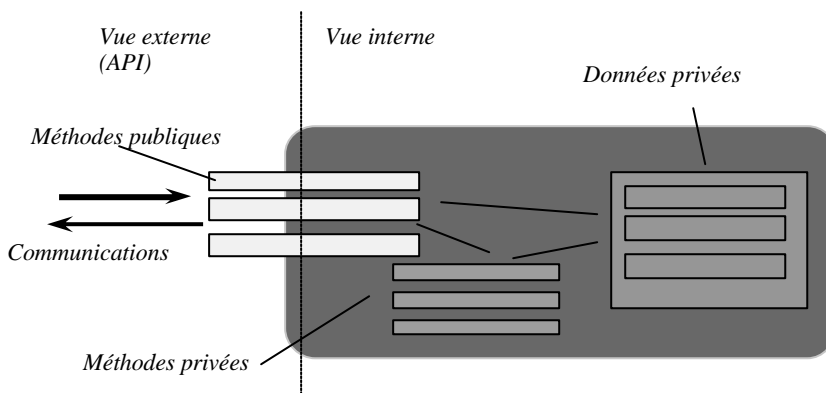
for (int i=0;i<v.size();i++)
    System.out.println(v.elementAt(i));

c1 = v.firstElement();           // Erreur de compilation, types incompatibles
c1 = (Compte) v.firstElement(); // correct

for (int i=0;i<v.size();i++)     // depose 10 à tous les comptes...
    ((Compte) v.elementAt(i)).deposer(10);
```

Encapsulation

- ❖ Définit une notion d'intérieur/extérieur
 - Tout ce qui est "interne" est invisible/inaccessible depuis l'extérieur (sauf depuis des objets de la même "famille")
- ❖ Un objet est une "cellule" (analogie biologique)



Les packages

- ❖ Constitue une unité de programme, pour structurer logiquement et physiquement des bibliothèques de classes.
 - Permet à la JVM de localiser les classes à charger
- ❖ Toute classe est définie dans un package
- ❖ Un package porte un nom:
 <package racine>.<sous package₁>...<sous package_n>
- ❖ Le nom complet d'une classe est donné par:
 <nom package>.<nom de classe>

Utilisation des packages

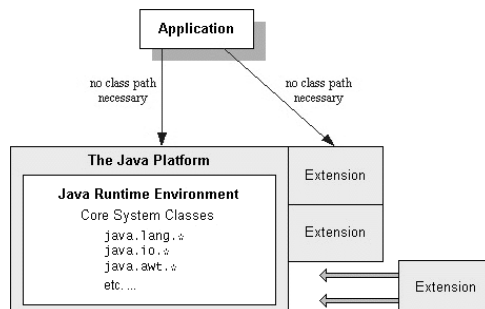
- ❖ Pour importer un package
 - import <nom de package>.<nom classe>
 - import <nom de package>.*
 - signifie que les classes importées peuvent être utilisées sans le nom de package
 - sinon on peut toujours utiliser le nom complet de la classe sans faire import
- ❖ Pour définir un nouveau package
 - package <nom de package>au début du fichier contenant la classe

Processus de chargement des classes

- ❖ Les classes sont chargées dynamiquement
 - à partir d'un emplacement dans le système de fichier ou à partir d'une archive .jar (ou .zip)
 - Le nom de package correspond à un directory
- ❖ Le CLASSPATH est une variable système qui contient l'ensemble des emplacements (racines de noms) où peuvent se trouver les classes à charger
- ❖ La JVM a besoin d'une classe
 - 1. Regarde si la classe est déjà dans la JVM
 - 2. Recherche s'il y a un CLASSPATH
 - Si oui, utilise ce classpath comme racine de tous les lieux où peut se trouver la classe
 - Si non, chercher dans le dossier extension un jar contenant la classe.
 - 3. si pas trouvée, aller rechercher dans le jar contenant toute la bibliothèque des classes initiales.
 - 4. si pas trouvée, erreur 'Class not found'
- ❖ Il est possible de (re)définir ses propres class loader pour aller chercher les classes où on le désire

Chargement de classes

- ❖ Le processus de chargement de classes est totalement indépendant de la compilation!!
- ❖ Java prend la première classe qu'il trouve
- ❖ le mécanisme d'extension a été défini pour qu'on puisse se passer de ce \$#@£&! de CLASSPATH
 - Mais à utiliser avec circonspection



L'accessibilité

- ❖ Détermine la visibilité d'une classe, d'une méthode ou d'un attribut par rapport aux autres classes et aux packages
 - private: accessible uniquement depuis la classe
 - " " (rien): accessible depuis le package
 - protected: accessible depuis le package + toutes les sous-classes de la classe
 - public : accessible depuis partout
- ❖ Le compilateur réclame qu'il n'y ait au plus une classe 'public' par fichier
- ❖ Attention:
 - les constructeurs doivent être public pour pouvoir instancier une classe 'public'

Les autres "modifieurs"

- ❖ Les modifieurs donnent des indications sur la nature d'une classe, attribut et/ou méthode
- ❖ static : informations associées à une classe
- ❖ final : indications de non redéfinition
- ❖ transient : indication de lien faible pour le GC
- ❖ + abstract et les indicateurs d'accessibilité (déjà vus)

static

- ❖ Permet d'associer des attributs et des méthodes à une classe
- ❖ Une méthode statique ne peut accéder qu'à des attributs et méthodes statiques

```
class Truc {
    static int n=0;
    int a;

    Truc(){
        n++;
    }

    static int getN(){return n;}
    static void test(){a=35;} // Erreur

    public static void main(String[] args){
        Truc t = new Truc(); // OK
    }
}
```

```
....
Truc t = new Truc();
Truc.getN(); // OK
t.getN(); // Possible
```

final

- ❖ Sert à indiquer qu'une classe, attribut ou méthode ne peut pas être redéfinie:
 - Pour une classe: pas de sous-classes possibles
 - Pour une méthode: pas de redéfinition de la méthode dans les sous-classes
 - Pour un attribut: pas de modification de la valeur de l'attribut ? constante
 - On utilise souvent le `public static final <type> <var> = <expr>` pour définir des constantes accessibles de plusieurs parties du programme.

```
class Truc {
    public static final int UNECONSTANTE = 34;
}

...
a = Truc.UNECONSTANTE * 2;
```

Les exceptions

- ❖ Mécanisme standard de récupération d'erreurs
- ❖ Les exceptions forment une hiérarchie de classe
- ❖ Possibilité de définir de nouvelles classes d'exception
 - issues de java.lang.Exception
- ❖ Le filtrage des exceptions utilise l'héritage

```
try {  
    ...  
}  
catch (<classe d'exception> <variable>){  
    ...  
}  
catch (...){  
    ...  
}
```

```
int i = 0;  
  
try {  
    int a = 100/i;  
}  
catch(ArithmeticException e) {  
    System.out.println("on a une division par zéro, mais on continue");  
}  
catch (Exception e){  
    System.err.println("erreur: "+e.getMessage());  
    e.printStackTrace();  
}
```

Les exceptions (2)

- ❖ Pour être interceptée, une exception doit être
 - Contenue dans le bloc try
 - Être située dans une méthode propageant (throws) cette classe d'exception
- ❖ Création d'une exception comme sous-classe de Exception
 - primitive throw pour lancer l'exception

Les exceptions (3)

```
class MonException extends Exception
  monException(String msg){
    super(msg);
  }
}
```

```
class Test {
  void p(int x) throws monException {
    if (x < 0)
      throw new MonException("valeur invalide");
    else
      .....
  }
}
```

```
class Toto {
  void utilise(int a){
    Test t = new Test();
    try {
      t.p(a);
    } catch(MonException ex){
      ...
    }
  }
}
```

classes anonymes

- ❖ Possibilité de créer une classe anonyme et de l'instancier en même temps (analogue aux lambda-expressions en Lisp)
- ❖ Très utilisé dans le cadre de la gestion d'événements (voir plus loin) et donc l'utilisation de composants graphiques
- ❖ Permet de créer une classe issue d'une interface
- ❖ Attention: les variables locales externes à l'objet doivent être "final" (pas de modifications des variables fermées). Mais les attributs ne posent pas de problèmes

```
interface Truc {
  void m();
}
```

```
class Chose {
  int b=80;
  void p(final int a){
    Truc t = new Truc(){
      int c = a;
      void m(){
        super.m();
        c++;
        b++;
      };
    };
    t.m();
  }
}
```