Chapitre 9 : Programmation Orientée Objets

Notion d'objet
Conception par objets
Interfaces, Classes
Création d'objets
Membres de classe
Package
Spécificité java 1.5

Notion d'objet

définition

caractéristiques

exemple

Notion d'objet

- Un objet est une unité de structuration du logiciel.
- Un objet :
  - possède un état (sa mémoire propre) constitué de valeurs (ses données).
  - possède des actions qui peuvent agir sur cet état pour éventuellement le modifier
  - contrôle ce qu'un client (utilisateur) peut voir et utiliser.

- L'intérêt du concept d'objet est double :
  - structurer le logiciel
  - assurer la protection contre d'éventuelles erreurs

Caractéristiques d'un objet

- **encapsule** données (attributs) et méthodes (fonctions qui déterminent son comportement). Données et méthodes sont par conséquent intimement liées au sein de l'objet.

- présente l'ensemble des services qu'il peut rendre sous la forme d'une **interface**. En fait, un objet est une structure de programme qui sépare le quoi du comment, le visible de l'invisible. On parle **d'abstraction de données** car il sépare les deux aspects :
  - **client**, l'interface contenant une vue abstraite des services qu'il peut rendre est rendue visible
  - **serveur**, la manière dont les services sont rendus est tenue secrète.

- possède la propriété de **masquer l'information** ("information hiding"). Ceci signifie que bien que tout objet puisse communiquer avec d'autres au travers d'interfaces bien définies, ils ne sont pour autant pas autorisés à savoir comment ceux-ci sont implantés. Les détails d'implantation sont cachés au sein des objets eux-mêmes. Une modification d'implantation n'a pas d'effet sur le programme.

Exemple d'objet

Un magnétoscope possède les caractéristiques d'un objet :

- un magnétoscope possède une structure propre. Il résulte d'un assemblage de composants (**ses données**) spécifiques,
- l'utilisateur peut interagir avec lui au moyen de boutons qui constituent son **interface** et remplissent des fonctions précises (**ses méthodes**),
- un magnétoscope **encapsule** au sein de sa structure données et méthodes,
- l'utilisateur n'a pas besoin de connaître le fonctionnement interne, ni la technologie sous-jacente (**masquage de l'information**),
- il peut être connecté à un autre objet tel qu'un récepteur TV ou un camescope pour former un système électronique.

Conception par objets

introduction

notion de modèle

méthodologies

exemple

Les paradigmes de la programmation

- La **programmation procédurale** envisage un programme comme une suite de commandes passées à une machine (Pascal, Fortran, basic,…)

- La **programmation fonctionnelle** place la notion de fonction au centre de la programmation. Un programme est vu comme une fonction qui met en correspondance un ensemble de données et un ensemble de résultats.
  (Lisp, Scheme, ML, …)

- Dans la **programmation par objets**, Un programme est conçu comme une sorte de modèle de la réalité (du domaine de l'application) (Ada)

- La **programmation orientée objets** (POO) propose la même vision avec des concepts plus avancés, en particulier les concepts d'héritage, d'interface et de polymorphisme. C++ et Java supportent l'ensemble de ces concepts.

Notion de modèle

- Le développement d'un système part d'une spécification imprécise, floue, avec éventuellement des ambiguïtés et/ou des inconsistances. Elle est exprimée en langue naturelle dans un cahier des charges.

Un modèle est une abstraction qui :

- présente le point de vue d'un observateur sur un aspect du monde réel (isolation de certaines propriétés d'un objet)

- est décrit dans un formalisme dont la syntaxe et la sémantique sont clairement définies

Méthodologie

En général, les méthodologies proposent trois niveaux de **raffinement** :

- **conceptuel**. C'est le niveau de l'analyse du système. Les entités modélisés sont les objets du monde réel, du domaine de l'application, leurs relations ont une sémantique propre,

- **logique**. C'est le niveau conception. Les entités modélisées sont des objets informatiques. Les objets sont liés par des relations dont la sémantique est prédéfinie,

- **réalisation**. C'est le niveau implantation. Les objets modélisés sont des structures du langage choisi. Les liens sont des structures du langage (pointeurs, références).

Conceptuel

Logique

Agenda
tableau:Page[]
noterRDV(String,Date) modifierRDV(String,Date) supprimerRDV(Date)

Réalisation

```
public class Agenda{
 Page[] tableau;
 public void noterRDV(String rdv,Date date){
       ....}
 public void modifierRDV(String rdv,Date date){
       ...}
 public void supprimerRDV(Date date){
       ...}
}
```

Méthodologies (2)

- SADT (décomposition dirigée par les traitements)
- SA/RT, spécifique aux systèmes temps réel (décomposition dirigée par les traitements)
- Merise, spécifique aux systèmes d'informations (décomposition dirigée par les données)
- OMT, méthodologie orientée objet
- Booch, méthodologie orientée objet
- UML, [UML 97] fusion de 3 méthodologies existantes et tentative de normalisation (http://www.rational.com/uml/index.jsp)

Interfaces

spécification d'objet

exemple

étude de cas

Spécification d'objet

• La spécification d'un objet peut être vu comme un **contrat** (une interface) qui lie l'utilisateur du composant et celui qui est chargé de l'implanter.

• Un contrat spécifie les opérations que l'on peut effectuer sur les données d'un objet. Il définit les services que peut rendre l'objet. Mais comme il s'agit d'une abstraction, il ne définit pas comment sont réalisées les opérations ni comment sont représentées les données.

• Le concept d'`interface` Java permet l'expression d'un tel contrat. Il définit en fait de manière abstraite un nouveau type de données. On parle de **Type Abstrait de Données**.

• L'utilisateur peut utiliser ces opérations dans son algorithme en respectant simplement le contrat, c'est à dire le type des opérations. La correction de l'algorithme peut alors être contrôlée (contrôle de type).

• Le développeur du composant a pour mission de garantir le contrat exprimé dans `l'interface` du composant lors de son implantation.

Comment spécifier une opération

- Spécification d'une opération dans une interface. Elle est constituée de 3 parties :
  - **signature** des opérations : spécification du type de chacune des opérations
  - **préconditions** : spécifier les conditions d'erreur pour chaque opération
  - **sémantique** : décrire la signification de chacune des opérations

```
/**
 * sémantique
 *   insère un élément e dans un tableau t à l'indice i
 * @param e l'élément inséré dans le tableau
 * @param i indice d'insertion de e dans le tableau
 * preconditions
 *    inserer(t,e,i) => borneInf(t) <= i <= borneSup(t)
 */
public void inserer(Elément e,int i);
```

Spécification d'objet : interface

- Tout utilisateur d'un Langage de Programmation (LP) utilise en fait plusieurs Types Abstraits de Données dont une implantation est prédéfinie dans le langage. Il s'agit des types primitifs du langage.
- Spécifier un objet revient à spécifier le Type Abstrait de Données auquel il appartient
- La notion d'interface Java permet de décrire un nouveau Type Abstrait de Données

Interface du type `Entier` (1/3)

L'interface du type `Entier` avec les opérations arithmétiques classiques :

• l'implantation garantira que la somme de deux entiers est un entier compris dans les limites des entiers représentables dans la machine (un mécanisme d'exception sera déclenché sinon).

• l'utilisateur aura la garantie, sans avoir à contrôler le code de l'addition (par exemple), que le résultat est bien un entier que son algorithme peut utiliser en tant que tel.

Interface du type `Entier` (2/3)

```
interface  IEntier{
 /**
  * sémantique : addition de 2 entiers
  * @param i la première opérande
  * @param j la seconde opérande
  * @return le résultat de l'addition
  */
 public IEntier plus(IEntier i,IEntier j);

 /**
  * sémantique : soustraction de 2 entiers
  * @param i la première opérande
  * @param j la seconde opérande
  * @return le résultat de la soustraction
  */
 public IEntier moins(IEntier i,IEntier j);
```

Interface du type `Entier`(3/3)

```
  /**
   * <b>sémantique : multiplication de 2 entiers
   * @param i la première opérande
   * @param j la seconde opérande
   * @return le résultat de la multiplication
   */
  public IEntier mult(IEntier i,IEntier j);

  /**
   * sémantique : division de 2 entiers
   * @param i le numérateur
   * @param j le dénominateur
   * @return le quotient de la division
   * preconditions
   *    div(i,j) => j≠0
   */
  public IEntier div(IEntier i,IEntier j);
}
```

Documentation générée par javadoc

interface **IEntier**

Author:
danielenselme

Method Summary

IEntier	**div**(IEntier i, IEntier j) sémantique : division de 2 entiers
IEntier	**moins**(IEntier i, IEntier j) sémantique : soustraction de 2 entiers
IEntier	**mult**(IEntier i, IEntier j) sémantique : multiplication de 2 entiers
IEntier	**plus**(IEntier i, IEntier j) sémantique : addition de 2 entiers

Method Detail

div

```
IEntier div(IEntier i,
            IEntier j)
```

sémantique : division de 2 entiers

Parameters:
i - le numérateur
j - le dénominateur
Returns:
le quotient de la division preconditions div(i,j) ? j?0

moins

```
IEntier moins(IEntier i,
              IEntier j)
```

sémantique : soustraction de 2 entiers

Parameters:
i - la première opérande
j - la seconde opérande
Returns:
le résultat de la soustraction

Etude de cas

On souhaite enregistrer à partir d'un capteur une série de mesures comme :

- la vitesse des véhicules sur une route pendant une période déterminée
- l'évolution de la température d'un solide au cours du temps

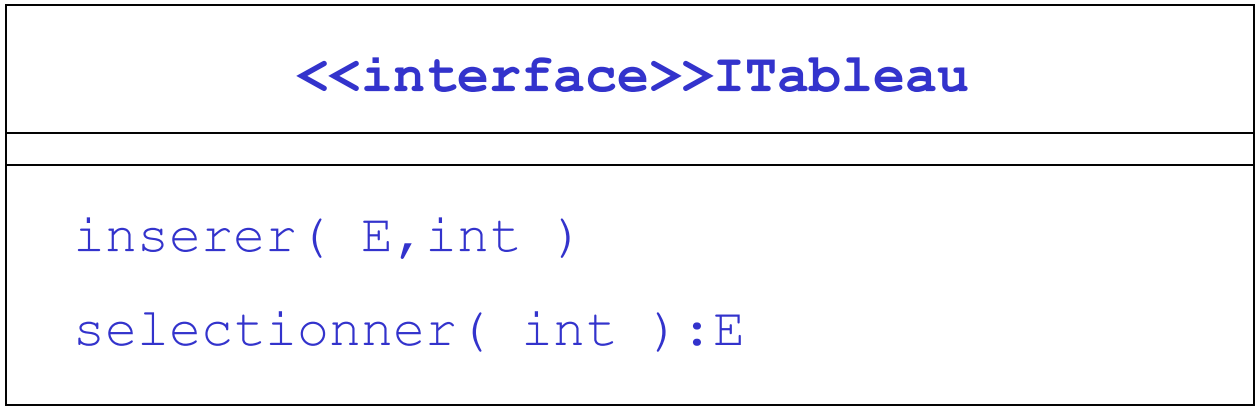
Un objet représentant cette série de mesures pourrait être un tableau de réels dont chaque élément est indexé par un entier. Les entiers représentent les tranches de temps et les réels représentent les différentes mesures.

Le même objet tableau pourrait être utilisé à d'autres desseins comme la représentation et la manipulation :

- des commandes à livrer pour une semaine donnée,
- des indices boursiers pour un mois donné.

Représentation UML

UML est un formalisme de modélisation qui regroupe un ensemble de règles pour représenter graphiquement interfaces, objets, classes et les différentes relations entre eux.



Interface ITableau (1/2)

```
interface  ITableau<E>{
 /**
  * sémantique
  *   insère l'élément e dans le tableau à l'indice i
  * @param e l'élément à insérer
  * @param i l'indice d'insertion
  * préconditions
  *  inserer(t,e,i) => borneInf(t) ≤ i ≤ borneSup(t)
  * pour pouvoir enregistrer l'élément e à l'indice i
  * de t, i doit appartenir à l'intervalle d'indices du
  * tableau
  */
 public void inserer(E e,IEntier i);
```

Interface `ITableau` (2/2)

```
/**
 * sémantique
 *   sélectionne un élément situé à l'indice i du
 * tableau
 * @param i l'indice de sélection
 * @return l'élément sélectionné
 * préconditions
 * selectionner(t,i) => defini(t,i)
 *   pour retrouver l'élément enregistré à l'indice i
 *   du tableau, il faut qu'un élément ait été au
 *   préalable enregistré.
 */
public E selectionner(IEntier i);
}
```

Classes

notion de classe

définir les variables d'instance

définir les méthodes d'instance

exemple

Notion de classe

Les classes sont des plans de construction pour un ensemble d'objets.
Une classe est une sorte de modèle pour décrire une collection d'objets.
Différentes classes décrivent différents ensembles d'objets.

Un objet est une instance (un exemplaire) d'une classe

Toute classe définit :

- les données caractéristiques de ses objets.
  Ce sont ses variables d'instance. Chaque objet d'une classe possède une copie de l'ensemble des variables d'instance avec des valeurs spécifiques qui forme l'état de l'objet.

- l'ensemble des actions que l'on peut effectuer sur les objets de la classe. Elles sont appelées méthodes. Elles agissent sur l'état de l'objet concerné.

Représentation UML d'une classe

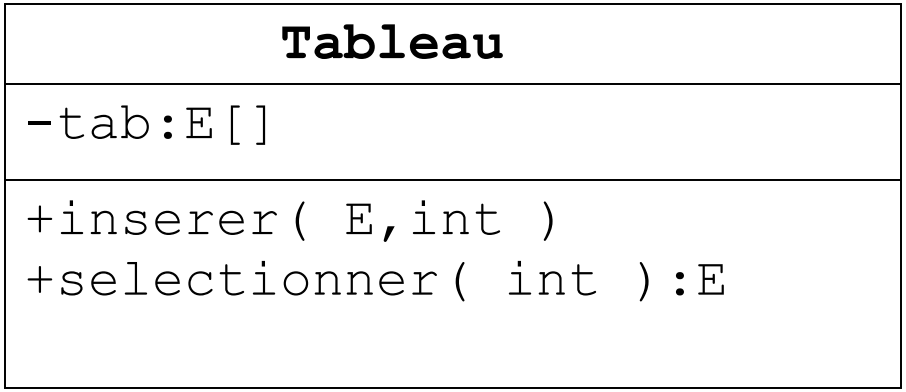
La spécification d'un objet est décrite par une interface.

L'interface est implantée par une classe.

remarque : en Java, l'interface peut être décrite explicitement ou bien contenue dans la définition de la classe

Exemple: la classe `Tableau` implante l'interface `ITableau`

les signes - et + indiquent les droits d'accès aux membres de la classe



Généralités sur les opérations

On distingue 3 sortes d'opérateurs :

- les **constructeurs** qui construisent physiquement un objet du type

- les **accesseurs** qui permettent la restitution d'une partie d'une valeur du type

  (`selectionner(t,i) -> e`),

- les **transformateurs** qui transforment une valeur du type (qui modifient l'état de l'objet)

  (`inserer(t,e,i)`).

On distingue aussi les opérations :

- **publiques**, accessibles à un client du type (signe +)
- **privées**, cachées à tout utilisateur du type (signe -)

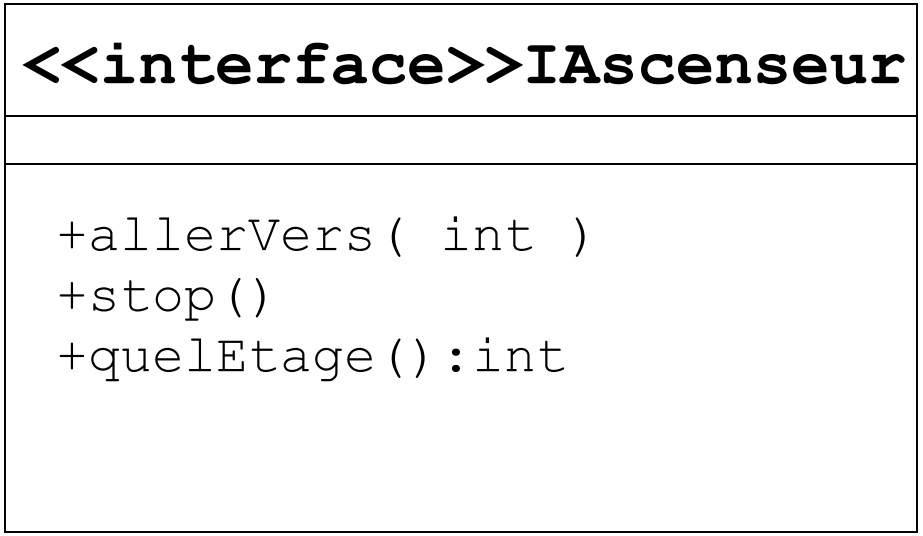
Modélisation d'un ascenseur

Dans le monde réel, un ascenseur peut-être décrit par

- ses caractéristiques : poids max, étage inférieur et supérieur, matériaux et technologie utilisés, ...
- ses fonctionnalités, aller d'un étage à l'autre, s'arrêter, indiquer l'étage courant, ...

La modélisation consiste à s'abstraire de certaines de ses caractéristiques et fonctionnalités de manière à concevoir un objet idéal qui réponde au cahier des charges de l'application.
La modélisation revient à choisir les attributs et les opérations pertinentes par rapport aux besoins de l'application

Représentation UML de l'interface `IAscenseur`



Interface `IAscenseur` (1/2)

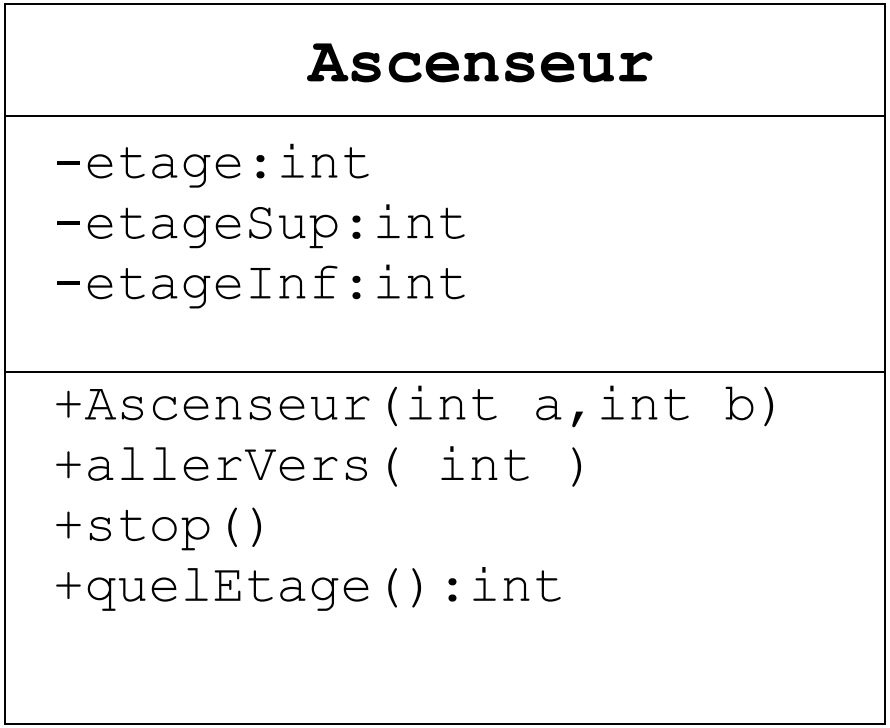
```
interface  IAscenseur{
 /**
  * sémantique
  *   déplace l'ascenseur vers l'étage i
  * @param i l'étage d'arrivée
  * préconditions
  * allerVers(a,i) => etageInf(a) ≤ i ≤ etageSup(a)

  */
 public void allerVers(int i);
 /**
  * sémantique
  *   arrête l'ascenseur
  */
 public void stop();
```

Interface `IAscenseur` (2/2)

```
 /**
   * sémantique
   *   retourne le n° de l'étage courant
   * @return le n° de l'étage
   */
 public int quelEtage();
}
```

Représentation UML de la classe `Ascenseur`



Implantation Java (1/2)

```
public class Ascenseur implements IAscenseur{
  // variables d'instance
  private int etage;
  private int etageInf = 0;
  private int etageSup = 8;
  // constructeurs
  public Ascenseur(){}
  public Ascenseur( int a,int b )
  {if ( a<b )
       {etageInf = a;etageSup = b;}
   else //erreur
  }
```

Implantation Java (2/2)

```
// méthodes
  public void allerVers( int v )
  {if ( etageInf <= v <= etageSup )
      etage = v;
   else //erreur
  }
  public void stop(){}
  public int quelEtage(){ return etage; }
}
```

Classe `Rectangle`

```
public class Rectangle{
  private double largeur = 10.0,hauteur = 5.0;
  public double surface(){
      return largeur*hauteur; }
  public double perimetre(){
      return 2*(largeur+hauteur); }
  ...
}
```

Classe `Complexe`(1/2)

```
public class Complexe{
 private float r = 0.0F;
 private float i = 0.0F;
 public void add( Complexe c ){
      this.i = this.i+c.i;
      this.r = this.r+c.r;
 }
 public void sub( Complexe c ){
      this.i = this.i-c.i;
      this.r = this.r-c.r;
 }
 public void possible(Complexe c){
      c.i=this.i+c.i;
 }
.....
}
```

`this` est l'objet sur lequel l'opération est effectuée

Classe `Complexe`(2/2)

```
public class Client{
      public static void main(String[] args){
      Complexe x = new Complexe();
      Complexe y = new Complexe();
      x.add(y);
      System.out.print("partie réelle de x=");
      System.out.println(x.r); //accesseur nécessaire
      x.i=90.1f; //erreur,transformateur nécessaire
    }
}
```

On ajoute les méthodes suivantes à la classe `Complexe`

```
  public float getR(){return r;}
  public void setR(float x){this.r=x;}
```

Classe `Compte`

```
public class Compte{
  private int solde = 0;
  public void crediter( int n ){
      this.solde = this.solde + n; }
  public void debiter( int n ){
      this.solde = this.solde - n; }
  public int combien(){
      return solde; }
}
```

Conversions (1/2)

type primitif -> Classe Wrapper
Classe Wrapper -> type primitif

```
Integer x   = new Integer(5);
int i       = x.intValue();
Float y     = new Float(3.14);
float j     = y.floatValue();
long z      = new Long(354);
long k      = z.longValue();
Double t    = new Double(3.14);
double l    = t.doubleValue();
Boolean b   = new Boolean(true);
boolean v   = b.booleanValue();
```

Conversions (2/2)

type primitif -> String

```
String s;
int i         = 5;
s             = String.valueOf(i);
boolean b     = true;
s             = String.valueOf(b);
float f       = 5.8;
s             = String.valueOf(f);
double d      = 43.89;
s             = String.valueOf(d);
boolean b     = false;
s             = String.valueOf(b);
```

Création d'objets

mécanisme de création d'objet

désignation d'un objet

envoi de message

Création d'objets

- Un objet (rectangle, compte bancaire ou nombre complexe) est crée (**instancié**) à partir d'une classe.

- Chaque classe détient donc un "plan de construction" pour des objets. Une classe décrit une famille d'objets qui possèdent **même structure et même comportement.**

- Toute classe possède la description de la structure de chaque objet qu'elle est capable d'engendrer. Elle décrit l'ensemble des **variables d'instance** (identificateur, type et éventuellement valeur initiale) et des **méthodes** (signature et code d'implantation).

Mécanisme de création d'un objet

- La création d'un objet se fait en 2 étapes :
  - **Construction** : c'est à dire allocation de mémoire pour l'objet par l'opérateur **`new`** appliqué au constructeur de la classe.

```
new Rectangle();
new Compte();
new Complexe();
```

  - **Initialisation** par appel d'un constructeur. Les paramètres du constructeur sont les initialisateurs de variables d'instance de l'objet. Dans le cas où aucun paramètre n'est spécifié une valeur par défaut est fournie par le langage (Java, C++).

```
new Rectangle( 0.0,0.0 );
new Compte( 700 );
new Complexe( 3.0F,1.25F );
```

Mécanisme de désignation d'un objet

- L'objet crée, pour être manipulé dans un programme, doit porté un nom (identificateur).
  - **Déclaration** d'une variable :

```
Rectangle unRectangle;
Compte leMien;
Complexe c;
```

  - **Association** de la variable à l'objet :

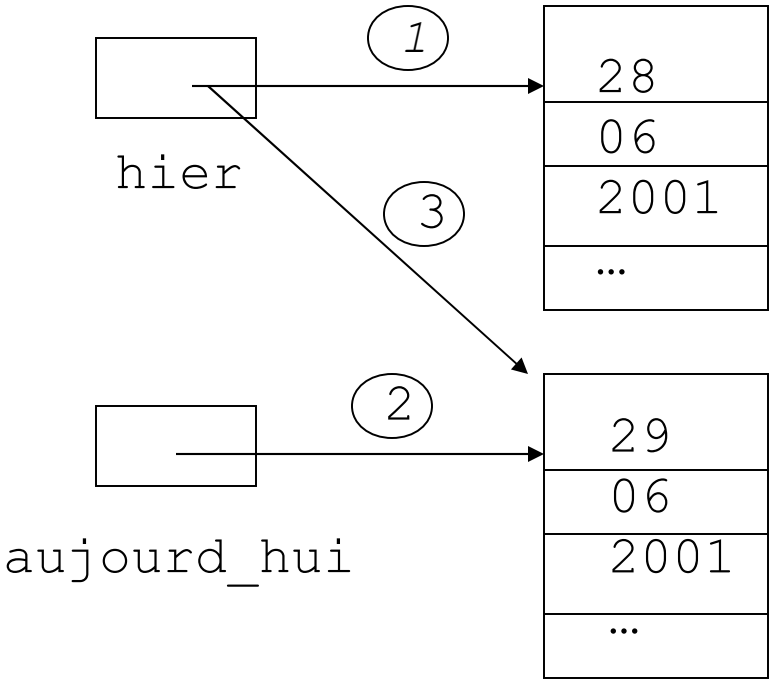
```
unRectangle = new Rectangle( 3.0,2.0 );
leMien = new Compte( 700 );
c = new Complexe( 5.0F,1.25F );
```

Copie

- Les variables Java contiennent des références à des objets et non les objets eux-mêmes. Lorsque l'on réalise une copie (par affectation), on copie en fait les références d'objets.

```
Jour hier = new Jour(28,06,2001); // 1
Jour aujourd_hui = new Jour(29,06,2001); // 2
hier = aujourd_hui; // 3
```

Représentation graphique



Clonage

- La copie effective d'objets est réalisée par clonage. On utilise la méthode `clone()` de la classe `Object`.

```
Jour demain = new Jour(30,06,2001);
hier = (Jour)demain.clone();
```

- La méthode `clone()` retourne un objet de la classe `Object`, une conversion explicite de type est, par conséquent, requise.

Envoi de messages

- Les objets **communiquent** entre eux par **envoi de messages**. Un objet émetteur demande à un objet récepteur de réaliser un traitement (une opération).

- L'objet récepteur peut voir son état modifié à l'issue de l'exécution du message.

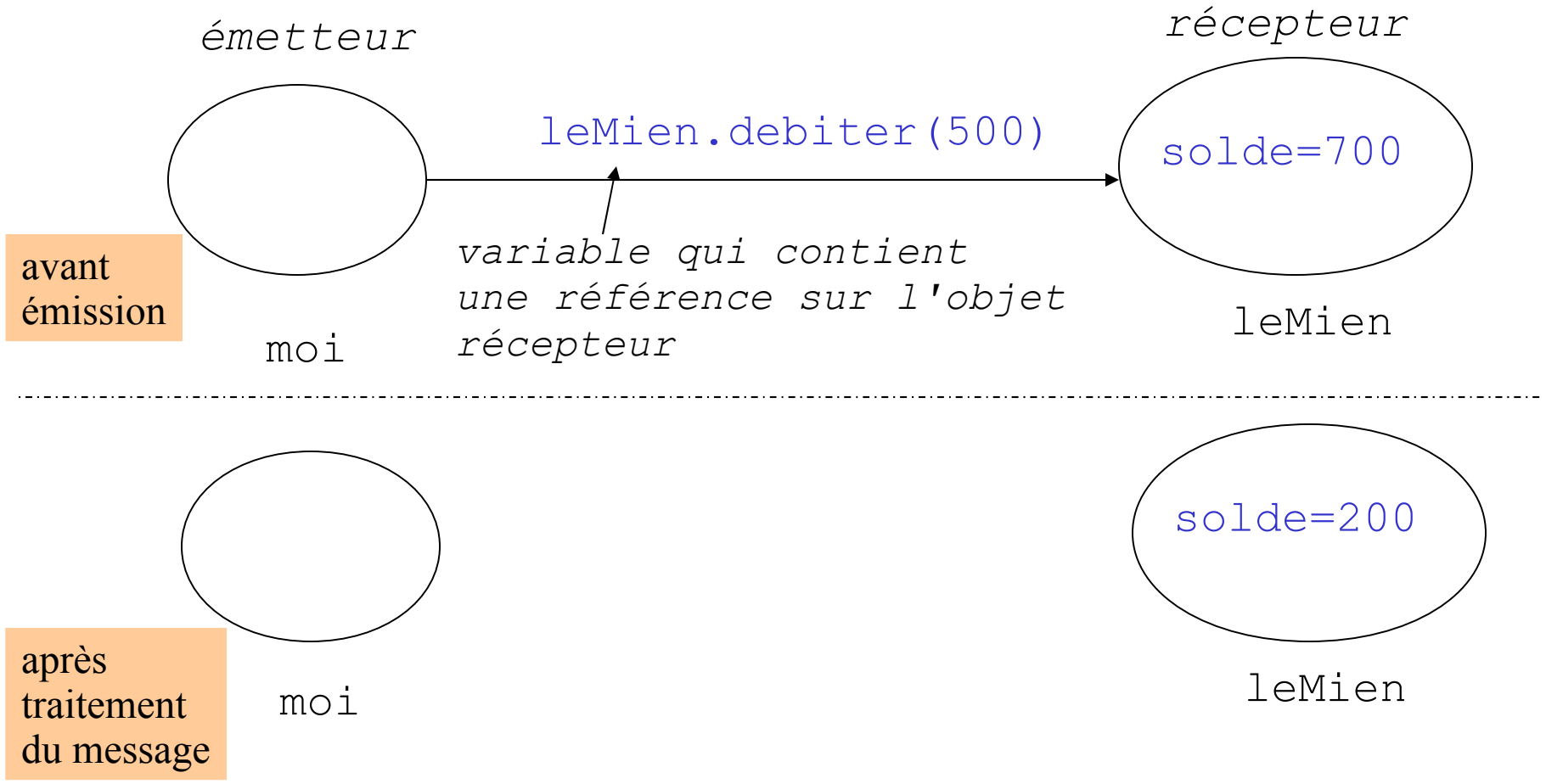
Structure d'un message

```
recepteur.sélecteur(arguments)
```

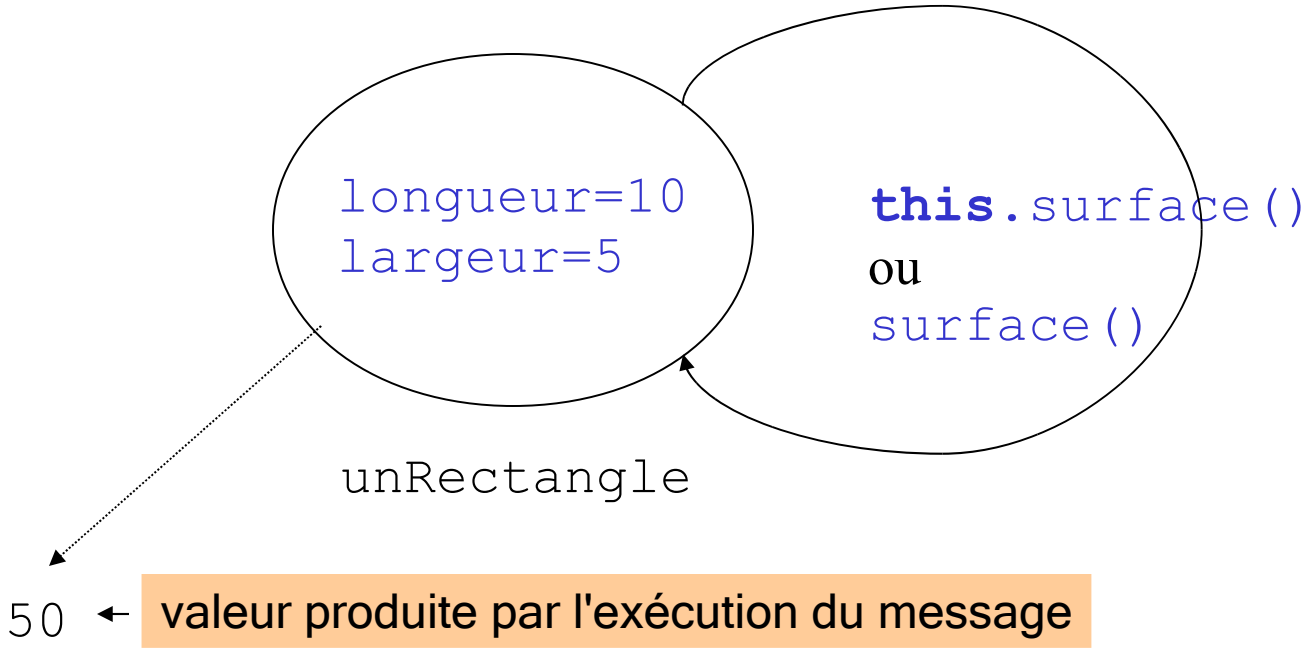
Exemples :

```
leMien.debiter( 500 );
// l'état de l'objet leMien a changé :
// le solde vaut 200
c.add( 2.F,2.F );
// l'état de l'objet c a changé :
// r vaut 5.F et i vaut 3.25F
unRectangle.surface();
// le message retourne la valeur 6.0
```

Envoi de message : exemple 1



Envoi de message : exemple 2



Sémantique d'exécution d'un message

- Evaluation des paramètres effectifs de la méthode invoquée
- La valeur de chaque paramètre est copiée dans le paramètre formel correspondant (appel par valeur). Si les valeurs des arguments sont des références (cas des tableaux et des objets), ce sont celles-ci qui sont copiées et non les objets référencés. Si les valeurs sont d'un type primitif (`int, double, char, boolean, ...`), la valeur de l'argument est copiée.
- Le récepteur du message est un paramètre implicite. C'est un objet dont la référence est affectée à la pseudo-variable **`this`**.
- Les instructions qui forment le corps de la méthode sont exécutées. La structure de contrôle de base est maintenant l'envoi de message.
- L'exécution de la méthode se termine par l'instruction **`return`** ou bien par la dernière instruction (cas d'une méthode qui retourne une valeur du type `void`).
- La valeur retournée remplace le message. Dans le cas d'une méthode `void`, l'instruction qui suit l'envoi du message est exécutée.

Exercice 1

Un autoradio permet le pré-enregistrement de 5 stations. 5 boutons permettent d'activer l'écoute de la station pré-enregistrée choisie.

Une mollette permet d'activer l'écoute directement sur la station de votre choix en se positionnant sur la fréquence correspondante.

Une station est représentée par sa fréquence en Mhz.

Une instance d'autoradio est crée en fixant par défaut les pré-sélections.

1. Implanter ce modèle d'autoradio sous la forme d'une classe Java.
2. Ecrire un programme utilisateur de cette radio qui :
   - crée un autoradio avec les stations pré-enregistrées suivantes : 101.5, 87.9, 105.1, 95, 101.1.
   - écoute la station 105.1
   - puis écoute la station 93.5

Membres de classe

la pseudo variable `this`

visibilité

constantes

variables de classe

méthodes de classe

Visibilité

- Les composants logiciels déclarés **`public`** sont utilisables par n'importe quelle classe.
- Les membres **`private`** ne sont accessibles que dans la classe qui les définit.
- Si aucun modificateur **`public`** ou **`private`** n'est spécifié, le membre d'un package (classe, méthode, variable) est accessible à toutes les méthodes du même package.

Protection des variables d'instance

- Il est de bon usage de déclarer les variables d'instances **`private`** et non **`public`**.
- La bonne approche est de fournir des **méthodes d'accès** à ces variables d'instance et de les garder **cachées**.
- Les méthodes d'accès sont alors les seules portes d'accès à ces variables.
- La correction, le déboggage et la maintenance du programme s'en trouvent améliorés.

Surcharge de constructeurs

```
public class Jour{
   int leJour;
   int leMois;
   int l_an;
   public Jour(){}

   public Jour( int j,int m,int a ){
        leJour = j;leMois = m;l_an = a;}

   public Jour( int m,int a ){
        leMois = m;l_an = a;}
   ...
}
```

La référence d'objet `this`

```
public class Jour{
  int j; int m; int a;
  public Jour(){}
  public Jour( int j,int m,int a ){
       this.j = j;this.m = m;this.a = a; }

  public Jour( int m,int a ){
       this.m = m;this.a = a; }
  ...
  public void println(){
       System.out.println( this );
       // réalise this.toString()
  }
}
```

Autre utilisation de `this`

```
public class Jour{
  int j;int m;int a;
  public Jour( int j ){
       this( j,getMois(),getAn());
       // getMois et getAn() retournent une valeur
       // pour m et a
  }

  public Jour( int m, int a ){
       this.m = m;this.a = a;
   }

  public Jour( int j,int m, int a ){
       this.j = j;this.m = m;this.a = a;
   }
  ...
}
```

Exercice 2

```
public class Exercice2 {
   int var = 0;
   public static void main( String[] args ){
     Exercice2 obj = new Exercice4();
     obj.methode();
   }
   public Exercice2(){ this( 15 ); }
   public Exercice2( int var ){ this.var = var; }
   void methode(){
     System.out.println( "var d'instance var=" + var );
   }
}
// Quel est le résultat affiché ?
```

Exercice 2 : résultat

```
var d'instance var=15
```

Exercice 3

```
public class Exercice3 {
  public static void main( String[] args ){
    ClasseA objA = new ClasseA();
    ClasseB objB = new ClasseB();
    objA.enregistrer( objB );objB.rappeler();objA.println();
  }}
class ClasseA{
  int var;
  void enregistrer( ClasseB obj ){ obj.enregistrer(this); }
  void rappeler( int var ){ this.var = var; }
  void println(){
    System.out.println( "variable d'instance var = " + var );
  }}
class ClasseB{
  ClasseA ref;
  void enregistrer( ClasseA var ){ ref = var; }
  void rappeler(){ ref.rappeler( 15 ); }}
```

Exercice 3 : résultat

```
variable d'instance var = 15
```

Déclaration de constante

- Lorsqu'une variable d'instance représente une valeur constante pour la classe, on la déclarera **`public static final`**.

- Bien sûr, si la constante n'a d'intérêt que pour la classe, on omettra le mot clé **`public`**.
  - **`public`** signifie que tout objet peut y accéder
  - **`static`** signifie que cette valeur est partagée par toutes les instances de la classe
  - **`final`** signifie que sa valeur ne peut pas être modifiée

- Exemples :

```
public static final double PI = 3.14159265358;
static final double TABLE[] = {1.28,3.32,8.97,5.65};
final String msg = "OK";
```

- Traditionnellement, en Java, une constante est notée en majuscules.

Variables `static`

- Lorsqu'un objet est crée (une classe est instanciée), un champ spécifique à chaque variable d'instance est crée. Il existe donc autant d'exemplaires d'une même variable d'instance que d'instances d'une même classe.

- Cependant, il est possible qu'une seule variable soit partagée par toutes les instances de la classe. On appelle ce membre, une **variable de classe**. Elle est distinguée des variables d'instance par le mot clé **`static`**.

Exemple (1/2)

```
public class Introspection{
  public static int NB_INSTANCES = 0;
  public Introspection(){
      NB_INSTANCES++;
  }
  public static void println(){
      System.out.println
            ( "Nombre d'instances:" + NB_INSTANCES );
  }
}
```

Exemple (2/2)

```
public class Externe{
  public static void main(String[] args){
    Introspection i1 = new Introspection();
    Introspection i2 = new Introspection();
    Introspection i3 = new Introspection();
    Introspection.println();
  }
}
```

Exercice 4

```
public class Exercice4 {
 private int var = 0; private static int var_static = 0;
 public Exercice4(int var,int var_static){
  this.var = var; this.var_static = var_static;
 }
 void println( int var ){
  int var_static = 1;
  System.out.println("paramètre : var = " + var);
  System.out.println("var locale : var_static =" + var_static);
  System.out.println("variable d'instance : var=" + this.var);
  System.out.println("var classe :var_static="+this.var_static);
  }
  public static void main( String[] args ){
    Exercice4 obj = new Exercice4(5,10);obj.println(20); }}
```

Exercice 4 : résultats

```
paramètre var = 20
var locale var_static  = 1
variable d'instance var = 5
var de classe var_static = 10
```

Méthodes `static`

- Le receveur d'un message n'est pas obligatoirement une instance de la classe `Introspection` mais la classe elle-même.

- Le mot clé **`static`**, lorsqu'il se rapporte à une méthode, permet d'éviter d'avoir à créer un objet (à instancier une classe) pour utiliser ses méthodes.

Exemple :

```
double d = Math.pow( x,2 );

double x = Math.round( 6.6 );

int i = Integer.parseInt( "123" );
```

Spécificités Java1.5

importation `static`

types énumérés

Types énumérés (1/2)

- en Java 1.4.x et précédents, un type énuméré est défini comme un ensemble de constantes :

```
static final int ETAT_INITIAL  = 0;
static final int ETAT_MIN      = 1;
static final int ETAT_MAX      = 2;
static final int ETAT_INTER    = 3;
```

- En Java 1.5,

```
enum Etat{ETAT_INITIAL,ETAT_MIN,ETAT_MAX,ETAT_INTER};
```

- Cette déclaration génère automatiquement une classe qui implante les interfaces `Comparable` et `Serializable`
- Elle fournit les méthodes :
  `values()` qui retourne un tableau contenant les constantes de l'énumération
  `valueOf( String )` qui retourne l'`enum` correspondant à la `String`
  `equals()`, `toString()` et `compareTo()`

Types énumérés (2/2)

```
public class Menu{
 public enum Option{FICHIER,EDITION,OUTILS,FORMAT};
 public static void main( String[] args ){
   for( Option choix:Option.values() )
              System.out.println( choix );
```

autre utilisation :

```
   for( Option choix :Option.values() ){
    switch(choix){
      case FICHIER:System.out.println("Fichier");break;
      case EDITION:System.out.println("Edition");break;
      case OUTILS:System.out.println("Outils");break;
      case FORMAT:System.out.println("Format");break;
    }
}
```

Importation de membres `static` (1/2)

- Comment importer facilement et uniquement les membres d'une interface
- En effet certains membres (les constantes) sont déclarée static et d'autres (les méthodes) pas

Soit l'interface

```
package outils;
interface A{
    public static final double MAX = 50;
    public static final double MIN = -50;
    public int compter();
}
```

- En Java 1.4.x, pour utiliser la constante `MAX`, on doit implanter l'interface `A` et donc la méthode `compter()`

```
public class Classe implements A{
    ...
    double val = 2*MAX;
    ...
    public int compter(){ ... }}
```

Importation de membres `static` (2/2)

- En java 1.5, on a la possibilité de n'importer que les membres `static` d'une interface ou d'une classe

```
import static outils.A.*;
public class Classe {
    ...
    double val = 2*MAX;
// Inutile donc d'implanter la méthode compter()
}
```

- De même,

```
import static java.lang.Math.*;
import static java.lang.System.*;
```

permet d'écrire

`sin(x)` plutôt que `Math.sin(x)`

`out.println("Hello")` plutôt que `System.out.println("Hello");`

AutoBoxing/Unboxing

- Java 1.5 fournit une nouvelle facilité d'écriture

en **Java 1.4.x** et précédents, on écrirait

```
List l = new ArrayList();
l.add( 0,new Integer(34) );
int n = ((Integer)(l.get(0))).intValue();
```

en **Java 1.5**

```
List<Integer> l = new ArrayList<Integer>();
l.add( 0,34 );
int n = l.get( 0 );
```

Packages

notion de package
convention de nommage
package et répertoire
introduction d'une classe dans un package
utiliser une classe à partir d'un package

Notion de package

Un package Java est un groupe de classes

L'API Java est un ensemble de **`packages`** qui contiennent des classes et d'interfaces reliées selon un thème commun (outils réseau dans `java.net`, outils graphiques dans `java.awt`, ...).

Il existe un package par défaut défini implicitement regroupant toutes les classes d'un programme

On peut définir explicitement des packages regroupant une ou plusieurs classes du programme

Intérêt des packages

- Localiser les classes
  - associer un thème à un ensemble de classes permet de les localiser plus facilement
- Eviter les conflits de nom
  - on peut différencier 2 classes de noms identiques par le nom du package auquel elles appartiennent
- Faciliter la distribution du logiciel
  - il est plus facile de distribuer un package que des classes isolées
- Protéger les classes
  - il est possible de n'autoriser l'accès aux membres d'un package qu'aux classes qui le constituent

Convention de nommage

- Les packages s'emboîtent hiérarchiquement

  `java.lang.Math` signifie que `Math` est une classe du package `java.lang`

  et que le package `lang` appartient au package `java`

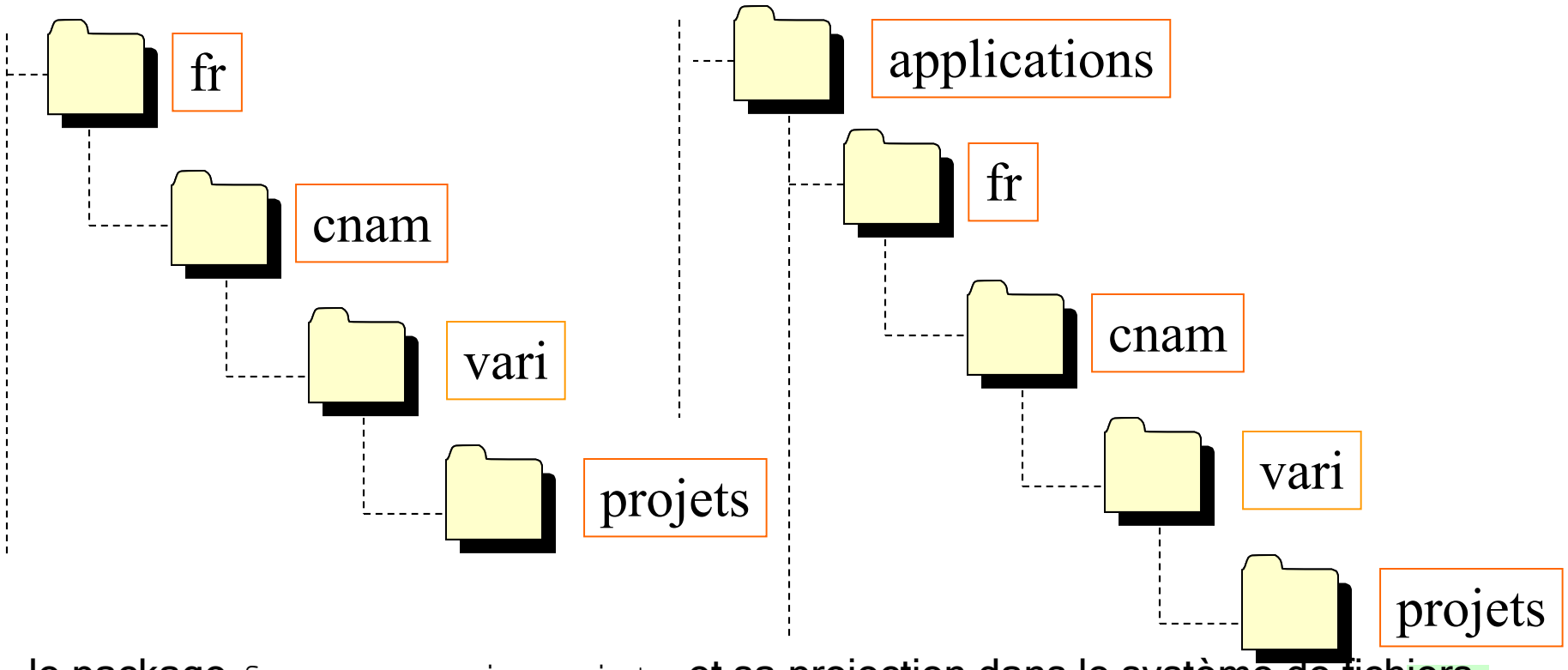
- Il est possible de nommer un package de manière unique sur l'Internet; pour cela, il faut débuter le nom du package par le nom du domaine de votre machine en ordre inverse.

```
package fr.cnam.vari.projets;
```

- Il y a une correspondance un à un entre le nom des packages et la structure du système de fichiers. Pour le package `fr.cnam.vari.projets`, il faut créer la hiérarchie de répertoires correspondante.

- Un package est un répertoire qui contient le bytecode de ses classes

Exemple (1/2)



le package `fr.cnam.vari.projets` et sa projection dans le système de fichiers

Exemple (2/2)

- Le répertoire applications n'est pas nécessairement la racine de l'arborescence des fichiers.
- Pour que Java connaisse l'emplacement d'un package dans le système de fichiers, il faut modifier la variable d'environnement `classpath`

  `classpath=.;c:\applications;`

  où `.` désigne le répertoire courant
- Les classes du package `fr.cnam.vari.projets` ainsi que les classes appartenant au répertoire courant sont alors utilisables dans le programme

Importation de package (1/2)

- La clause **`import`** suivie du nom du package en début de fichier programme permet l'utilisation de toutes les classes contenues dans le package.

```
import fr.cnam.vari.projets.*;
```

Ou bien une seule classe

```
import java.util.Scanner;
```

Importation de package 2/2)

Le compilateur utilise un objet `classLoader` pour localiser les classes dont il a besoin.

Sa recherche commence par les classes standards du J2SDK

En cas d'échec, il continue la recherche par le `classpath`, ce qui permet d'accéder à une liste d'archives (`.jar ou .zip`)

Les noms uniques sur l'internet permettent d'éviter les conflits de noms

Visibilité

– Un package peut être implanté sur plusieurs fichiers

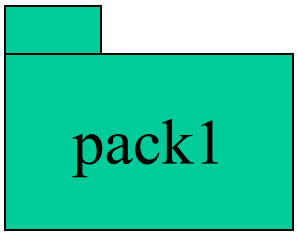
- Un fichier peut contenir plusieurs classes.

- A chaque classe du package déclarée **`public`** correspond un fichier `.java`.

– On ne trouve donc qu'une classe **`public`** par fichier.

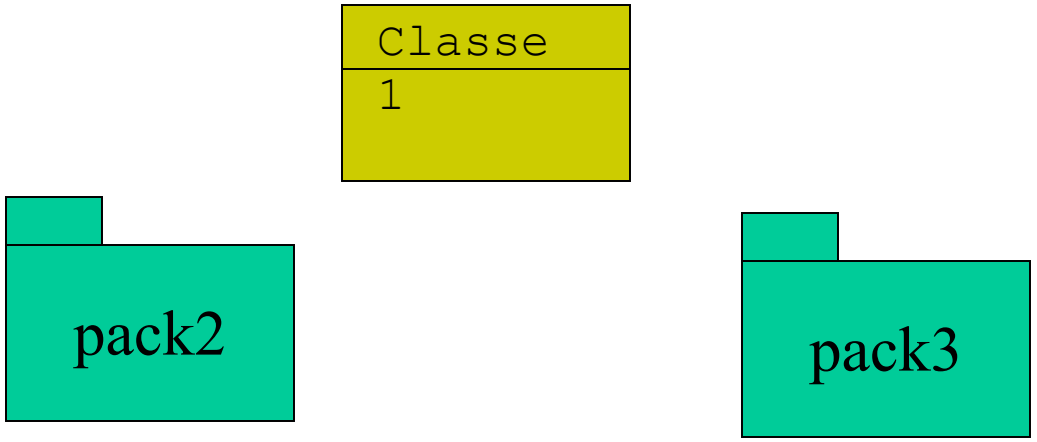
– Les autres classes du fichier sont alors invisibles aux classes extérieures au package

Package (1/6)



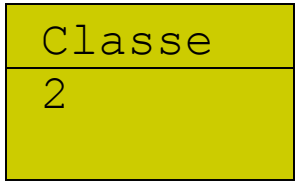
Package (2/6)

Contenu du package `pack1`

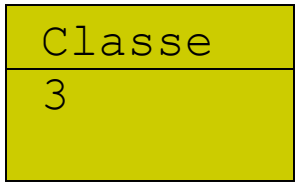


Package (3/6)

contenu du package `pack2`



contenu du package `pack3`



Package (4/6)

```
package pack1;

import pack1.pack3.*;
import pack1.pack2.Classe2;

class Classe1{
  public static void main( String[] args )
  {
    System.out.println( "Démarrage du pack1" );
    System.out.println( "Instancie des objets de différentes
   classes dans différents packages" );
    new Classe2();
    new Classe3();
    System.out.println( "Retour au pack1" );
  }
}
```

Package (5/6)

```
package pack1.pack2;
public class Classe2 {
  public Classe2(){
    System.out.println
       ( "\t-->objet du pack2 sous-package du pack1" );
  }
}

package pack1.pack3;
public class Classe3 {
  public Classe3(){
    System.out.println
       ( "\t-->objet du pack3 sous-package de pack1" );
  }
}
```

Package (6/6) : résultat affiché

```
Démarrage du pack1

Instancie des objets de différentes classes dans
   différents packages

   -->objet du pack2 sous-package du pack1

   -->objet du pack3 sous-package de pack1

Retour au pack1
```

Les fichiers

Introduction

- les données d'un programme sont perdues lorsque le programme termine.
- pour rendre des données persistantes, il faut les sauvegarder sur disque ou CD, ...
- les données sont de type varié : type primitif, tableaux, objets
- elles ont généralement une structure complexe, hiérarchique (classe java)
- elles sont enregistrées en tant que "record" (enregistrement) dans un fichier

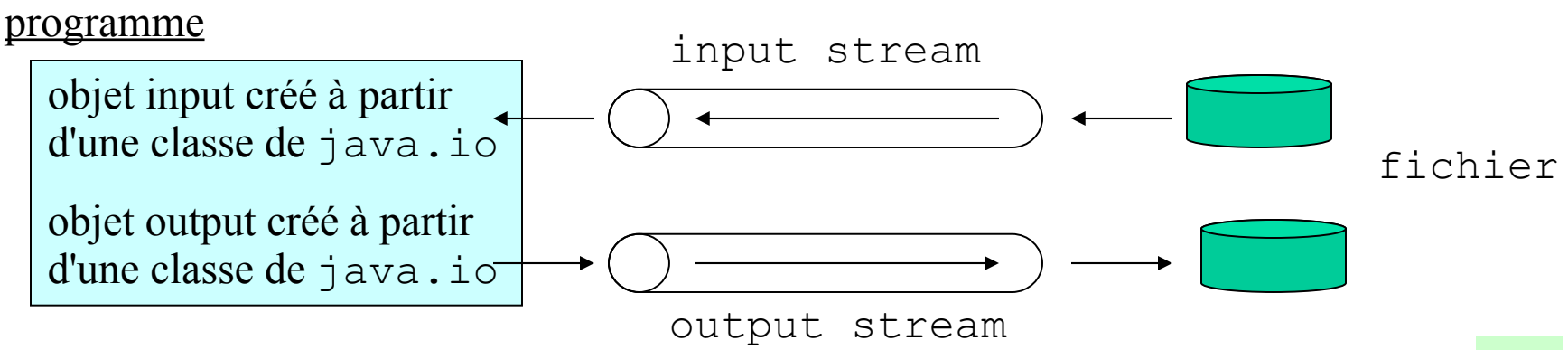
Fichier et "stream"

- Un fichier est un groupe d'enregistrements ("records") reliés entre eux
- Les entrées et sorties Java sont basées sur le concept de "stream" ou flot de données
- Java voit chaque fichier comme un flot séquentiel de bytes
- Chaque OS fournit un mécanisme pour détecter la fin de fichier
- Tout programme Java qui traite un fichier reçoit une indication de l'OS quand le programme atteint la fin du fichier
- Java fournit de nombreuses classes pour lire et écrire sur fichier décrites dans le package `java.io`

Programme java et "stream"

- L'ouverture d'un fichier par le programme provoque la création d'un objet auquel est associé un "stream"

- Par défaut tout programme Java ouvre 3 fichiers en créant 3 objets :
  - le fichier clavier vu à travers l'objet stream `System.in`
  - le fichier écran vu à travers l'objet stream `System.out`
  - le fichier erreur (messages sur écran) vu à travers l'objet stream `System.err`



La classe `File`

- La classe `File` permet de spécifier un fichier dont on connaît le nom relatif, absolu ou son URI (Unified Resource Identifier)

```
new File("monFichier.text")
new File("C:/projets/monFichier.text")
new File("file:///C:/projets/monFichier.text")
```

- Elle n'offre pas la possibilité d'ouvrir, ou de traiter le fichier

- Pour lire/écrire un fichier, il est nécessaire de créer un objet à partir des classes du package `java.io`

```
FileWriter output = new FileWriter(new File("temp.txt"));
```

Fichiers textes et fichiers binaires

- Dans un **fichier texte**, chaque octet (**byte)** est interprété comme un simple caractère codé en ASCII. On peut donc afficher de manière lisible ces fichiers grâce à n'importe quel éditeur de textes. A l'intérieur du fichier, certaines séquences de caractères sont utilisées pour encoder le formatage du texte (exemple : balises HTML).

- Un **fichier binaire** serait complètement illisible puisque chaque byte serait interprété comme un caractère ASCII, alors qu'il s'agit d'une simple séquence de bits dont la signification dépend de l'interprétation qui en est faite. La seule chose qu'ils aient en commun est qu'ils stockent des données sous forme de 0 et de 1. Il existe une grande quantité de fichiers binaires : fichiers images, sons, programmes, video, ...

Lecture à partir de fichiers texte

Lire un fichier ou lire les données tapées sur un clavier sont deux opérations équivalentes. On importe la classe `java.util.Scanner` :

- Lecture à partir du fichier texte `monFichier.txt`. Les enregistrements lus sont séparés par des espaces

```
Scanner in = new Scanner( new File( "monFichier.txt" ) );
String s = in.next();
```

ou bien

```
int i = in.nextInt();
```

ou encore

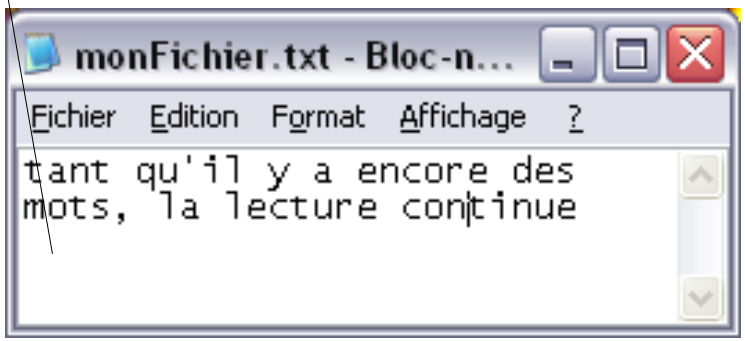
```
boolean b = in.nextBoolean();
```

- Lecture à partir clavier

```
Scanner in = new Scanner( System.in );
double d = in.nextDouble();
```

Exemple

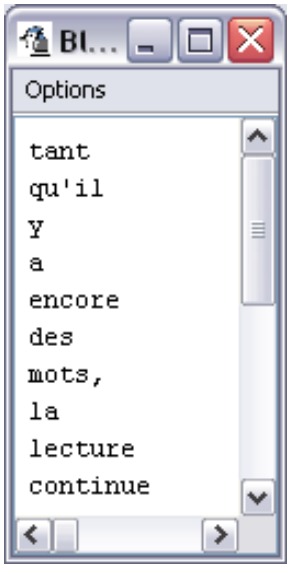
Fin de fichier marquée par `crtl d`




```
import java.util.*;
import static java.lang.System.*;
import java.io.*;

public class Texte{
  public static void main(String[] args)
                                          throws FileNotFoundException{
    Scanner in = new Scanner( new File("monFichier.txt") );
    String s = null;
    do{
        s = in.next();
        out.println(s);
    }while(in.hasNext());
    in.close();
  }
}
```

```
FileNotFoundException
NoSuchElementException
```



Fermeture d'un fichier

- Si le programme termine normalement, le système ferme automatiquement le fichier.

Mais

- En cas de terminaison anormale, le fichier restera ouvert et pourra être endommagé. Il est donc préférable de le fermer dès que possible au cours de l'exécution.

- Si le programme écrit dans le fichier puis plus tard le lit, alors il faut le fermer avant sa relecture. Aucun fichier ne peut être ouvert à la fois en lecture et en écriture en java.

Lecture de fichiers texte avec `BufferedReader`

- Création d'un flot en lecture

```
BufferedReader in =
   new BufferedReader( new FileReader("data.text") );
```

- La méthode `readLine()` permet de lire une ligne d'un texte et non pas des éléments séparés

```
String ligne = in.readLine();
```

- Pour tester la fin d'un fichier, on teste le contenu de la ligne lue à `null`

```
while ( ligne!= null )
```

Lecture de fichiers texte avec `BufferedReader`

Compter le nombre de lignes d'un fichier texte

```
BufferedReader in =
   new BufferedReader( new FileReader("data.text") );
int nbLignes = 0;
String ligne = in.readLine();
while( ligne!=null ){
   nbLignes++;
   ligne = in.readLine();
}
System.out.println("nombre de lignes = "+nbLignes);
```

Ecriture dans un fichier texte

- La classe `PrintWriter` possède les méthodes `print()` et `println()` utilisées pour écrire sur un terminal. Elle permet d'écrire des données variées (`String`, `int`,`double`,`float`,`boolean`, ...)

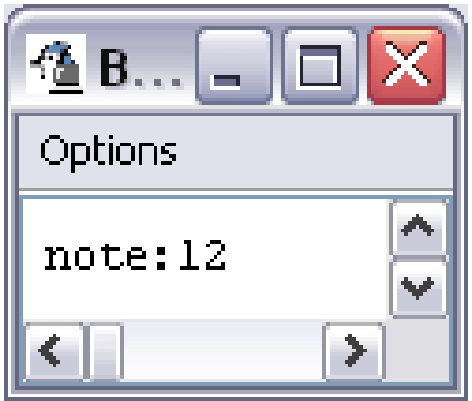
```
PrintWriter out = new PrintWriter("monFichier.txt");
out.println( "Ceci est le nouveau contenu du fichier" );
out.close();
```

- La méthode `println()` ci-dessus se comporte sur un fichier texte exactement comme la méthode `println()` dans `System.out.println();`

Exemple

```
import java.util.*; import java.io.*;
import static java.lang.System.*;
public class Texte{
    public static void main(String[] args)
                throws FileNotFoundException,IOException{
        String s = null;
        PrintWriter output = new PrintWriter("temp.txt");
        output.print("note:  "); output.print(12);
        output.close();
        Scanner in = new Scanner( new File("temp.txt") );
        s = in.next();
        int i = in.nextInt();
        out.println(s+i);
        in.close();
    }
}
```

Espace = séparateur des données



Fichiers binaires

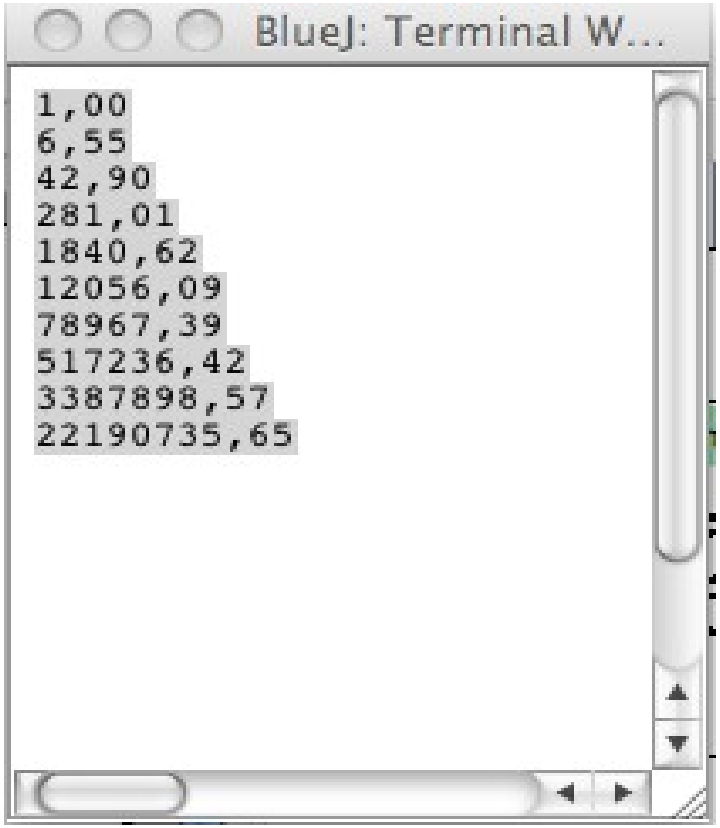
- Aucune information de typage n'est enregistrée dans un fichier binaire. Seul le programme détermine le type de la donnée enregistrée.
- La lecture d'un fichier binaire doit donc être totalement compatible avec la séquence enregistrée.
- Deux classes sont disponibles pour effectuer des E/S sur des fichiers binaires : `DataInputStream` et `DataOutputStream`

Les classes `DataOutputStream`, `DataInputStream`

`DataInputStream`	`DataOutputStream`
`readBoolean()`	`writeBoolean()`
`readByte()`	`writeByte()`
`readChar()`	`writeChar()`
`readDouble()`	`writeDouble()`
`readFloat()`	`writeFloat()`
`readInt()`	`writeInt()`
`readLong()`	`writeLong()`
`readShort()`	`writeShort()`
`readUTF()`	`writeUTF()`

```
import java.io.*; import java.text.*;
public class Binaire{
  public static void main( String[] args )
                    throws FileNotFoundException,IOException{
    FileOutputStream fos = new FileOutputStream( "data.bin" );
    DataOutputStream dos = new DataOutputStream( fos );
    int nb = 10;double x = 1.0;
    dos.writeInt( nb );
    for( int i = 1;i <= nb;i++ ){
       dos.writeDouble( x );x = x*6.55;
    }
    dos.close();
    FileInputStream fis = new FileInputStream( "data.bin" );
    DataInputStream dis = new DataInputStream( fis );
    nb = dis.readInt();
    for( int i = 1;i <= nb;i++ ){
        NumberFormat formatter = new DecimalFormat("#.00");
       System.out.println(formatter.format(dis.readDouble()));
    }
   }
}
```

Résultat


BlueJ: Terminal W...
1,00
6,55
42,90
281,01
1840,62
12056,09
78967,39
517236,42
3387898,57
22190735,65