Base de données et programmation avancée

Cours Java

Référence :	IIIS/Cours Java/PhF
Version :	1.2
Date:	02/08/2000

Version	Pages modifiées	Objet de la mise à jour
1.0	-	Création
1.1	-	Mise à jour 1999. Ordre des §exceptions, E/S et IHM modifié.
1.2		• Invocations des méthodes et attributs. • Ajout de l'exercice « Attributs représentatifs d'objets » • Les objets graphiques Java Swing • § Threads déplacé dans cours JDBC • La gestion des E/S • Les impressions.

SOMMAIRE

EXERCICES

1. Présentation du langage

Les objectifs de ce chapitre sont :

1. Connaître l'origine du langage
2. Connaître les caractéristiques essentielles de Java
3. Connaître les différents composants de J.D.K.
4. Connaître les principes de l'approche objet

1.1 Origine du langage : enjeu commercial

1. Le développement du réseau Internet mettant en liaison des machines hétérogènes,
2. Les coûts de déploiement de chaque nouvelle version sur un parc de machines étendu.

Sont des atouts considérables pour un langage indépendant de la plateforme d'exploitation.

1.2 Caractéristiques essentielles de Java

1.2.1 Simplicité

1. Simplicité par rapport au C++,
2. Réduire les risques de bugs par la suppression,
   - Des pointeurs
   - Allocation / Désallocation de mémoire (garbage collector)

1.2.2 Objet et distribué

1. Java reprend les caractéristiques des différents langages objets (Syntaxe du C++),
2. Toute ligne de code se trouve dans une classe (très fortement objet, plus que C++)
3. Java est un langage intermédiaire entre un langage compilé et un langage interprété par la génération et le téléchargement sur la machine virtuelle d'un pseudo code.

1.2.3 Robuste

1. Le compilateur Java est très strict
2. Contrôle de typage fort
3. Les exceptions et les erreurs retournées doivent être traitées
4. La gestion de la mémoire n'est plus à la charge du développeur,
5. Contrôle des débordements d'index.

1.2.4 Portable

1. Le compilateur crée un pseudo code interprétable par une machine virtuelle, (c'est la machine virtuelle qui doit être portée ! ! !)
2. Le pseudo code est conçu pour être rapidement interprété, optimisable au moment de l'exécution

1.2.5 Sécurisé

Java est conçu pour réaliser des logiciels distribués :

- Une applet peut se connecter et travailler seulement sur le serveur depuis lequel elle a été chargée,
- Impossible d'accéder aux ressources de la machine client (disques, socket …)
- Impossible de se connecter à un autre serveur.

1.2.6 Multi-thread

Java permet le développement multi tâches. La Machine Virtuelle supporte le Multi Threading:

- Echange de données entre thread,
- Synchrônisation de thread.

1.2.7 Performances

Technique utilisée	Machine Virtuelle	Just In Time Compiler	Compilateur
Vitesse de Java % C++	10 à 20 fois plus lent	3 à 5 fois plus lent	1.5 à 2.3 fois plus lent

1.3 Les différents composants du J.D.K (Java Development Kit)

La version du J.D.K (au 01/10/2000) est 1.3

Le J.D.K. contient :

- Un ensemble de classes de base regroupées en packages (System, Abstract Window Toolkit, Threads, JDBC…)
- Des exécutables : machine virtuelle, compilateur, outil de création de la documentation à partir du code.
- Le site de téléchargement est : http://www.javasoft.com/.

1.4 Les principes de l'approche objet

Java est un langage objet, pour l'utiliser il faut maîtriser les concepts de la modélisation objet.

La classe est une abstraction d'une entité du monde réel.

L'abstraction consiste à ne retenir de l'entité réelle que les caractéristiques nécessaires pour le système développé. Trop de détails rend le modèle complexe à gérer, pas assez de détails et le modèle ne décrit pas l'entité réelle correctement.

1.4.1 Intérêts de l'approche objet

L'approche objet permet **d'encapsuler** les traitements. Dans le cadre de développement d'une simulation du trafic routier, il est préférable que « Le moyen de transport » n'ait pas d'interaction néfaste avec « La route ».

Une classe offre une partie de ses méthodes (les publiques). Elle masque en général ses attributs et une partie de ses méthodes (les privées).

1.4.2 La classe.

Nom de la classe :
MoyenDeTransport
Attributs :
Couleur : string, Puissance : int, Cylindrée : float, Modèle : string.
Méthodes :
Contact(onOff), Demarrer() Accelerer() Freiner().

1.4.3 Une instance d'une classe .

MaVoiture **: MoyenDeTransport**
Attributs :
Couleur : « Blanche », Puissance : 45, Cylindrée : 945, Modèle : « 205 ».

1.4.4 Liens entre objets

Les objets collaborent entre eux. Pour cela un objet utilise les services d'un autre objet.

Nom de la classe :
FeuDeCirculation
Attributs :
Couleur : string, Période : int.
Méthodes :
Changer(Couleur)

Nom de la classe :
MoyenDeTransport
Attributs :
Couleur : string, Puissance : int, Cylindrée : float, Modèle : string.
Méthodes :
Contact(onOff), Demarrer() Accelerer() Freiner().

La méthode Changer de la classe FeuDeCirculation met en œuvre les méthodes Accelerer ou Freiner de la classe MoyenDeTransport.

1.4.5 La classification (héritage).

Nom de la classe :
MoyenDeTransport
Attributs :
Couleur : string, Puissance : int, Cylindrée : float, Modèle : string.
Méthodes :
Contact(onOff), Demarrer() Accelerer() Freiner().

Nom de la classe :
Camion
Attributs :
PTC : int,
Méthodes :

La classe « Camion» est une **spécialisation** de la classe « MoyenDeTransport ».

Réciproquement la classe « MoyenDeTransport » est la **généralisation** de la classe « Camion»

1.4.6 Le polymorphisme.

Une même méthode peut avoir plusieurs implémentations selon la spécialisation.

Par exemple : Accelerer, ne sera pas implémentée de la même façon selon que le moyen de transport est un camion, une moto ou une voiture. Cette méthode polymorphe sera redéfinie dans chacune de ces classes.

2. Elements de base

Les objectifs de ce chapitre sont :

1. Connaître les types de base
2. Connaître la syntaxe d'initialisation
3. Connaître les opérateurs.
4. Connaître les conversions de type
5. Connaître les structures de contrôle
6. Utiliser les commentaires

2.1 Les types de base

1. Les types arithmétiques :

byte	8 bits signés	Complément à 2
short	16 bits signés	Complément à 2
int	32 bits signés	Complément à 2
long	64 bits signés	Complément à 2
float	32 bits signés	Format IEEE
double	64 bits signés	Format IEEE

1. Le type caractère.

char	16 bits signés	Format UNICODE2

1. Le type booléen.

boolean	1 bit	2 valeurs possibles : true ou false

Note : Java ne connaît ni les pointeurs, ni les structures, ni les unions, ni les énumérés !

2.2 La syntaxe d'initialisation

La déclaration et l'initialisation d'une variable est de la forme

Type Nom-de-la-variable = Valeur-initiale ;

Le Nom-de-la-variable commence forcément par un caractère non numérique ou par un tiret bas (que l'on réserve aux attributs).

Attention :

char c = 'a' ; // la variable c est initialisée avec le caractère a

char c = 48 ; // correct, la variable c est initialisée avec le caractère dont le code UNICODE2 vaut 48 en décimal.

int v = 'a' ; // correct, la variable v est initialisée avec le code UNICODE2 du caractère a.

// Une chaîne de caractère String s'initialise avec des doubles côtes.

System.out.println (''hello'' + ' !') ; affiche hello33
System.out.println (''hello'' + ''!'') ; affiche hello !

Entiers :

par défaut les entiers sont exprimés en base 10. Ils peuvent être exprimés en octal en débutant par un 0, ou en hexadécimal en débutant par 0x :

29 035 0x1D 0X1d représentent tous la même valeur.

Par défaut un entier est codé sur un int à moins que l'on ne spécifie 29L pour un long.

Flottants :

18. 1.8 E 1 .18 E 2 représentent tous la même valeur.

Par défaut un flottant est codé sur un double à moins que l'on ne spécifie 18.0f pour un float.

2.3 la syntaxe d'initialisation de variables statiques

Les variables statiques sont initialisées au chargement de la classe sur la machine virtuelle.

```
class ListeElements {

        static String Elements [] = new String [5] ;

        static {
                Elements[0] = ''Premier élément'' ;
                Elements[1] = ''Deuxième élément'' ;
                Elements[2] = ''Troisième élément'' ;
                Elements[3] = ''Quatrième élément'' ;
                System.println.out (‘’ liste des éléments initialisée'') ;
        }
}
```

2.4 les opérateurs

Arithmétiques (sur entier ou flottant)

+, -, *, /, %

Unaires (sur entier ou flottant)

-, --, ++

Binaires bit à bit (sur entier)

~ inversion de tous les bits d'un entier,
& ET,
^ XOR,
| OU,
<< décalage à gauche, on remplit avec des 0
>> décalage à droite avec propagation du bit de signe
>>> décalage à droite, on remplit avec des 0

Comparaison (entiers, boolean, ou flottants)

==, !=, <, <=, >, >= ces opérateur retourent un boolean

Logique (sur boolean)

!, &&, ||

Conditionnel

? :

```
valeur = (condition ? valeurSiTrue : valeurSiFalse) ; est équivalent à
if (condition)
        valeur = valeurSiTrue ;
else
        valeur = valeurSiFalse ;
```

2.5 les opérations sur flottants

Java définit les valeurs infini positif et négatif.

x	y	x/y	x%y
valeur finie	+/- 0.0	+/- infini	NaN
valeur finie	+/- infini	+/- 0.0	x
+/- 0.0	+/- 0.0	NaN	NaN
+/- infini	valeur finie	+/- infini	NaN
+/- infini	+/- infini	NaN	NaN

les valeurs infinies sont définies par des constantes :
Float.POSITIVE_INFINITY ou
Double.NEGATIVE_INFINITY ou
Double.NaN

2.6 les conversions de type

4 modes de conversions :

```
int i, j ;
double f ;
short s ;
```

Conversion explicite :

```
i = (int) f ; // conversion explicite d'un flottant en entier par troncature.
```

Conversion par affectation :

```
s = i ; // attention aux risques de débordement.
```

Conversion par appel d'une méthode :

```
int obj.methode (int i) ;

obj.methode (s) // s est converti en entier à l'appel.
```

Conversion dans des calculs :

```
f = i / f // i sera converti en double avant d'effectuer la division.
```

Attention à la perte de valeur : float vers int, int vers short …

2.7 les structures de contrôle :

2.7.1 Les tests :

```
if ( condition_1 ) {
        instructions ;
} else if (condition_2) {
        instructions ;
} else if (condition_3) {
        instructions ;
} else {
        instructions ;
}
```

2.7.2 Les boucles :

- Boucle while : 2 types de boucles selon que le test a lieu au début de la boucle ou à la fin.

```
do {
        instructions ;
} while (condition) ;
```

ou

```
while (condition) {
        instructions ;
}
```

- Boucle for :

```
int i ;
for ( i=0 ; i < 6 ; i++) {
        instructions ;
}
```

ou

```
for (int i = 0 ; i < 6 ; i++) {
        instructions ;
}
```

2.7.3 Le commutateur multiple « switch, case, default »:

```
char c = (char) System.in.read () ;
switch (c) {
case 'o' :
case 'O' :
        instructions ;
        break ;
case 'n' :
case 'N' :
        instructions ;
        break ;
default :
        instructions ;
}
```

2.7.4 Les ruptures de séquence :

Lignes :

```
for (int l = 0 ; l < 10 ; l++) {
        System.out.println (''Ligne '' + l) ;
        Colonnes :
        for (int c = 0 ; c < 2 ; c++) {
                System.out.println (''Colonne '' + c) ;
                switch (valeur[l][c]) {
                case 4 :
                        break ; // ne fait rien dans ce cas. Va directement à
                        //l'affichage de la valeur
                case 0 :
                        continue Lignes ; // retourne au for l pour le l suivant
                default :
                        break Lignes ; // abandonne le for l
                }
                System.out.println (''Valeur '' + valeur[r][c]) ;
        }
}
```

2.8 Utilisation des commentaires:

/* ceci est un bloc de commentaire
sur plusieurs lignes */

// ceci est un commentaire sur une seule ligne, ou en fin de ligne.

/** ceci est un commentaire récupérable par javadoc pour la création d'une documentation au format Html */

3. Objets, Classes et Tableaux

Les objectifs de ce chapitre sont :

1. Connaître la syntaxe Java
2. Construire une classe
3. Déclarer les attributs d'une classe.
4. Déclarer les méthodes d'une classe,
5. Définir et écrire un constructeur,
6. Utiliser les références sur les objets
7. Manipuler les tableaux.

3.1 Syntaxe d'un fichier java:

La syntaxe de déclaration d'une classe et la suivante :

```
class NomDeLaClasse { // Début de la classe.

        // Définition des attributs
        private     int         _premierAttribut ;
        public      boolean     _deuxiemeAttribut ;

        // Définition des méthodes
        // Le constructeur
        NomDeLaClasse (int LeParametre) { // Début du constructeur
                instructions ;
        } // Fin du constructeur

        void PremiereMethode (int PremierParametre, float DeuxiemeParametre)
        {
                instructions ;
        }

        int     DeuxiemeMethode () { // Début de la méthode
                instructions ;
        } // Fin de la méthode
} // Fin de la classe.
```

3.2 Déclaration d'une classe:

La déclaration d'une classe suit la syntaxe suivante :

[final] [public] class Nom [extends NomSuperClasse] [implements Interface1à N] { …. }

où

- **final** la classe ne peut pas être étendue (par défaut elle peut être étendue).
- **public** la classe est visible en dehors du package (par défaut elle n'est pas visible).
- **extends** étend la super classe.
- **interface** interfaces implémentées par la classe.

3.3 Déclaration des attributs:

La déclaration d'un attribut suit la syntaxe suivante :

[final/static/transient] [public/protected/private] Type Nom ;

final l'attribut est constant, il doit être initialisé lors de la déclaration,
static une seule valeur de l'attribut pour toutes les instances de la classe,
transient ce type d'attribut ne sera pas sauvé lors de la sauvegarde de l'objet (utilisé dans les ejb)..

public l'attribut est visible depuis l'extérieur de la classe (visibilité par défaut si rien de spécifié),
protected l'attribut est visible depuis les classes qui étendent cette classe,
private l'attribut n'est visible que depuis la classe.

Type définit le type de l'attribut : type de base (int, short, char …), tableau ou référence à un objet.

3.4 Invocation des attributs d'instance:

Depuis la classe :

```
class Crayon {
        private String _couleur ;
        public String proprietaire ;
        private void affecterCouleur (String Couleur) {
                // l'attribut est invoqué directement par son nom.
                _ couleur  = Couleur.
        }
        private void affecterProprietaire (String proprietaire ) {
                // utilisation du this pour différencier l'attribut du paramètre
                this.proprietaire  = proprietaire ..
        }
}
```

Depuis une autre classe :

```
class Eleve {
        private Crayon _crayon ;
        private String nom ;
        private void acheterUnCrayon (Crayon UnCrayon) {
                _crayon = UnCrayon ;
                // l'objet point l'attribut qui doit être public
                _crayon.proprietaire  = nom ;
        }
}
```

3.5 Invocation des attributs de classe:

Depuis la classe ils sont accessibles comme les attributs d'instance

```
class Crayon {
        public static int nombreCrayons = 0;
        public Crayon () {
                // On crée un nouveau crayon donc le nombre de crayons croît
                nombreCrayons++ ;..
        }
}
```

Depuis une autre classe :

```
class Trousse {
        private void ajouterUnCrayon () {
                // l'objet point l'attribut qui doit être public
                Crayon crayon = new Crayon () ;
                // nombre de crayons créés dans l'application
                int nombreCrayons = Crayon.nombreCrayons;
        }
}
```

Exercice : Déclaration des attributs d'une classe (Vehicule_1)

Ecrire une classe Véhicule dont les caractéristiques sont : vitesse, direction, propriétaire

Exercice : Déclaration d'une variable statique (Vehicule_2)

Ajouter le numéro minéralogique du véhicule et un numéro d'ordre unique pour l'ensemble des véhicules.

3.6 Déclaration des méthodes:

3.6.1 syntaxe de la déclaration

[final/static] [public/protected/private] TypeRetour Nom ([Paramètres]);

final la méthode ne peut pas être surchargée dans une classe qui étend la classe.

static méthode de classe

qui ne peut utiliser que :

- des variables statiques de la classe,
- des méthodes statiques,

ne peut pas utiliser la référence this.

En général on encapsule des méthodes statiques dans une classe « utilitaires » utilisée par les classes qui en ont besoin.

public/protected/private : idem attribut.

3.7 Invocation des méthodes d'instance:

Depuis la classe :

```
class DisqueDur {
        public void lireDisqueDur () {
                // appel d'une méthode de la classe
                mettreEnRotation() ;
        }
        private void mettreEnRotation() {}
}
```

Depuis une autre classe :

```
class Ordinateur {
        private DisqueDur _disqueB ;
        private void demarrer () {
                // Appel d'une méthode d'instance
                _disqueB.lireDisqueDur() ;
        }
}
```

3.8 Invocation des méthodes de classe:

Depuis la classe elles sont accessibles comme les méthodes d'instance

```
class Utilitaires {
        public static float surfaceRectangle(float Longueur, float Largeur) {
                return Longueur * Largeur ;
        }

        public static float volumeRectangle (float Longueur, float Largeur, float
Hauteur)
        {
                float surface = surfaceRectangle(Longueur, Largeur) ;
                return surface * Hauteur ;
        }
}
```

Depuis une autre classe :

```
class Bassin {
        private float _largeur ;
        private float _longueur ;
        private float _profondeur ;
        private float volume () {
                return Utilitaires.volumeRectangle (_longueur , _largeur ,
_profondeur )
        }
}
```

3.8.1 passage des paramètres

Tous les paramètres sont passés par valeur. Quand le paramètre est une référence sur un objet **c'est la valeur de la référence qui est passée**.

Exemple	Résultat

public static void main (String[] args) {
double one = 1.0 ;
System.out.println (''avant : one = '' + one) ;
moitie (one) ;
System.out.println ('aprés : one = '' + one) ;
}
public static void moitie (double arg) {
arg /= 2.0 ;
System.out.println (''moitie : arg = '' + arg) ;
}
	avant : one = 1
	moitie : arg = 0.5
	après : one = 1

class Vehicule {
public String _proprietaire ;
public long _vitesse ;
public String toString () {
return (__proprietaire + _vitesse) ;
}
}
public static void main (String[] args) {
Vehicule voiture = new Vehicule (''Dupont'',120) ;
 	
System.out.println (''avant : voiture = '' + voiture) ;
effacer (voiture) ;
System.out.println ('aprés : voiture = '' + voiture) ;
}
public static void effacer (Vehicule) {
Vehicule._proprietaire = ''Nouveau proprietaire'' ;
Vehicule = null ;
}
	avant : voiture = Dupond 120
	
	après : voiture = Nouveau proprietaire 120

3.8.2 la surcharge

La surcharge (**overloading**) d'une méthode est autorisée.

C'est à dire que l'on peut définir plusieurs méthodes de même nom qui ne se différentient que par les types des paramètres.

Attention :

- Le paramètre de retour n'est pas pris en compte par le mécanisme de résolution de surcharge.
- Si l'appel ne correspond à aucune signature, java est capable via un algorithme de « moindre coût à la conversion » de choisir la méthode la plus proche.

3.8.3 le constructeur:

Le constructeur est la méthode appelée automatiquement à la création de l'objet.

- Son nom doit être identique à celui de la classe,
- Elle n'a pas de valeur de retour,
- Son rôle est d'initialiser les attributs de l'objet.
- Par défaut java génère automatiquement un constructeur qui initialise les attributs tel qu'ils sont définis dans la classe.
- Plusieurs constructeurs peuvent exister avec des signatures différentes.

3.8.4 le main:

Le main est la méthode appelée automatiquement au lancement de l'application.

- Cette méthode doit être public, static et void (elle ne retourne rien).
- Elle doit accepter en paramètre un tableau de Strings

3.8.5 la méthode toString():

La méthode toString() de n'importe quel objet est appelée automatiquement lorsque un objet de cette classe est utilisé dans une concaténation de chaîne de caractères :

System.out.println (''Ce que vaut l'objet : '' + objet) ;

Par défaut comme tout objet est une extension de la classe Object c'est la méthode toString() de la classe Object qui est utilisée si elle n'est pas surchargée.

Quand une référence nulle est convertie en une chaîne de caractères le résultat est ''null''.

Un constructeur peut en invoquer un autre :

```
class NomDeLaClasse { // Début de la classe.

        // Définition des attributs
        private     int         _premierAttribut ;
        public      boolean     _deuxiemeAttribut ;

        // Définition des méthodes
        // Un constructeur
        NomDeLaClasse (int LeParametre) { // Début du constructeur
                _premierAttribut = LeParametre;
        }

        // Un autre constructeur
        NomDeLaClasse ( int         LePremierParametre,
                        boolean     LeDeuxiemeParametre) {
                // Appel du 1er constructeur
                this (LePremierParametre) ;
                _deuxiemeAttribut = LeDeuxiemeParametre;
        }
} // Fin de la classe.
```

3.9 Création d'un objet

L'opérateur new permet de créé un nouvel objet.
Java :

- alloue l'espace mémoire nécessaire aux variables d'instances,
- appelle le constructeur,
- retourne une référence sur l'objet créé.

Il n'y a pas de destructeur, la mémoire est automatiquement libérée par la Garbage Collector lorsque qu'il n'existe plus aucune référence sur l'objet.

Le Garbage Collector est un Thread de la Machine Virtuelle. Il compacte la mémoire pour éviter la fragmentation.

3.10 Les références sur les objets

Les références sont des pointeurs sur des objets alloués dynamiquement par l'opérateur new.

3.10.1 this : référence sur l'objet courant.

```
class gare {
        private     String      _nom ;
        private     Liste       _listeAgents ;

        void EmbaucherAgent (Agent   NouvelAgent) {
                _listeAgents.add (NouvelAgent) ;
        }
}

class Agent {
        private     String      Nom ; // _nom serait préférable.
        private     Gare        _gareDAffectation ;

        Agent (Gare GareDAffectation, String Nom) {
                this.Nom = Nom ; // _nom = Nom serait préférable
                _gareDAffectation = GareDAffectation ;
                _gareDAffectation.EmbaucherAgent (this) ; // envoi de
l'adresse de l'objet courant à la méthode EmbaucherAgent de la classe Gare.
        }
}
```

3.10.2 opérateur sur les références

Opérateurs :

- == égalité
- != différence
- instanceof est une instance de.

null : permet d'affecter un pointeur. Retour d'une méthode …

3.10.3 visibilité

Contrairement au C++, le compilateur java gère les références vers des entités qui seront définies plus loin dans le fichier.

```
class Agent {

        Agent (Gare GareDAffectation, String Nom) {
                _nom = Nom ;
                _gareDAffectation = GareDAffectation ;
                _gareDAffectation.EmbaucherAgent (this) ;
        }

        private     String      _nom ;
        private     Gare        _gareDAffectation ;
}
```

Attention à la lisibilité du code !

Exercice : Constructeurs (Vehicule_3)

Ajouter à la classe véhicule un constructeur qui prend en paramètre le nom du propriétaire. Puis créer un 2ème constructeur qui prend en paramètre le nom du propriétaire et son n° minéralogique et utilise le premier constructeur. Ecrire une méthode qui affiche les caractéristiques du véhicule. Ecrire un programme de test qui crée 2 véhicules et affiche leurs caractéristiques.

Exercice : Déclaration et développement des méthodes (Vehicule_6)

Ajouter à la classe véhicule et mettre en oeuvre :

- Une méthode « lirePropriétaire » retournant le propriétaire du véhicule.
- Une méthode « modifierNuméroMinéralogique » permettant d'affecter un nouveau numéro minéralogique.

Exercice : Public/privé, définition des constantes (Vehicule_6)

Ajouter à la classe véhicule et mettre en oeuvre :

- Une méthode privée « tourner » qui prend en entrée un angle en flottant et ajoute cet angle à la direction courante du véhicule.
- Définir deux constantes : VIRAGE_GAUCHE et VIRAGE_DROIT ainsi que 2 méthodes publiques « tournerADroite », « tournerAGauche » qui utilisent la méthode « tourner » ainsi que les constantes pour faire virer de + ou - 90 degrés le véhicule.

Exercice : Méthodes statiques (Vehicule_6)

A partir du numéro d'ordre défini dans la classe Vehicule. Ajouter une méthode « nombreDeVéhicules » qui retourne le nombre de véhicules déjà créés, la mettre en oeuvre.

Exercice : Attributs représentatifs d'objets

Développer une classe FraisDeTransport

- les caractéristiques sont :
  - Longueur du trajet.
  - Prix du kilomètre
  - Frais de péage.
  - Frais de parking
- Les services sont :
  - Constructeur avec en paramètre le prix du kilomètre.
  - Ajouter (distance, frais de péage, frais de parking)
  - Totaliser () retourne le total des frais de transport

Développer une classe FraisDeSejour

- les caractéristiques sont :
  - Nombre de nuits.
  - Nombre de repas
  - Prix forfaitaire repas
  - Prix forfaitaire nuit
- Les services sont :
  - Constructeur avec en paramètre les prix forfaitaire repas et nuit.
  - AjouterJournee (le nombre de journées), augmente le nombre de nuits et de repas.
  - AjouterNuit (le nombre de nuits), augmente le nombre de nuits.
  - AjouterRepas (le nombre de repas), augmente le nombre de repas.
  - Totaliser () retourne le total des frais de séjour

Développer une classe NoteDeFrais

- les caractéristiques sont :
  - les frais de transport.
  - les frais de séjour.
  - Le mois
  - Le nom de l'employé
- Les services sont :
  - Constructeur avec en paramètre le mois, le nom de l'employé les prix forfaitaire repas et nuit ainsi que le prix du kilomètre
  - AjouterJournee (le nombre de journées), augmente le nombre de nuits et de repas.
  - AjouterNuit (le nombre de nuits), augmente le nombre de nuits.
  - AjouterRepas (le nombre de repas), augmente le nombre de repas.
  - AjouterFraisDeTransport (distance, frais de péage, frais de parking)
  - Totaliser retourne le total des frais de séjour et de transport

Depuis un programme principal créer une Note de frais. Activer les différentes méthodes de la classe. Afficher le total de la note de frais.

3.11 Les tableaux

3.11.1 Les tableaux de taille pré définie :

int TableauEntiers[ ] = new int [10] // tableau de 10 entiers, plutôt C++

Ou encore

int[ ] TableauEntiers = new int [10] // tableau de 10 entiers, plutôt Java

3.11.2 Les tableaux de taille paramétrable

```
int[ ] DefinitionTableauEntiers (int Dimension) {
        int [ ] TableauEntiers = new int[Dimension] ;
        return TableauEntiers ;
}
```

Il existe la méthode System.arraycopy (Object src, int srcPos, Object dst, int dstPos, int count) qui permet de copier un tableau dans un autre.

Exercice : Tableaux (Tableaux)

Déclarer un tableau d'entiers à 2 dimensions de 3 par 4. Initialiser le tableau à sa déclaration. Ecrire une méthode de création d'un tableau d'entiers à une dimension paramétrable, l'initialiser. Recopier le tableau déclarer dynamiquement dans une des dimensions du tableau d'entiers à 2 dimensions.

4. Classification, Packages, Encapsulation

Les objectifs de ce chapitre sont :

1. Mettre en œuvre l'héritage
2. Connaître les règles de conversion
3. Savoir utiliser le polymorphisme
4. Savoir utiliser les packages

4.1 Classification

4.1.1 Héritage

Java ne gère que **l'héritage simple.** C'est à dire qu'une classe ne peut hériter que d'une seule autre classe (en plus de la classe Object).

Par défaut toute classe est une sous classe de la classe Object.

Redéfinition (overriding) : la classe dérivée peut modifier l'implémentation d'une méthode ou de plusieurs méthodes.

Une classe ne peut pas hériter d'une classe définie « final » Cf. §3.2.

On ne peut pas modifier l'implémentation d'une méthode déclarée « final » Cf. §3.4

Exemple :

```
class Vehicule {
        private String          _marque ; // Non visible des classes qui étendent
la classe Vehicule
        private String          _numero ;
        private String          _proprietaire ;
        protected float         _prix // Visible depuis les classes qui étendent la
classe Vehicule.

        void   Vendre (String _NouveauProprietaire) {
                _proprietaire  = _NouveauProprietaire ;
                _prix = _prix * 0.9f ;
        }

        final String LireMarque () { return _marque} ; // Ne peut pas être dérivée

}
```

```
final class Voiture extends Vehicule { // La classe Voiture ne pourra pas être
étendue.
        private boolean         _peintureMetallisée ; // La classe Voiture a ses
attributs + ceux de la classe dont elle hérite.

        void Vendre (String _NouveauProprietaire) {
                _proprietaire  = _NouveauProprietaire ;
                _prix = _prix * 0.8 ;
        }

}
```

4.1.2 Conversion entre classe

Une variable qui référence un objet d'une classe peut référencer un objet de n'importe quelle de ses sous-classes.

class Vehicule {….}

final class Voiture **extends** Vehicule {….}.

Voiture voiture = new Voiture() ;
Vehicule vehicule ;

vehicule = voiture // OK toutes les voitures sont des véhicules
voiture = vehicule // KO tous les véhicules ne sont pas des voitures

4.1.3 Cast

vehicule = voiture // cast up

// cast down **DANGER** peut provoquer une exception à l'exécution.
voiture = (voiture) vehicule ;

4.1.4 Polymorphisme (overriding)

Une méthode polymorphique est une méthode déclarée dans une classe et redéfinie dans une sous classe.

Par défaut une méthode n'est pas déclarée « final » elle peut donc être redéfinie dans la classe qui en hérite.

On obtient ainsi plusieurs implémentations de la même méthode.

Attention :

Le choix de la **méthode** est guidé par le **vrai type** de l'objet.
Le choix d'un **attribut** est guidé par le **type déclaré** de l'objet

```
class SuperShow {
        public String str = ''SuperStr'' ;
        public void show () {System.out.println (''Super.show :'' + str) ;}
}


class ExtendShow extends SuperShow {
        public String str = ''ExtendStr'' ;
        public void show () {System.out.println (''Extend.show :'' + str) ;}
        public void main (String[] args) {
                ExtendShow ext = new ExtendShow() ;
                SuperShow sup = ext ; // cast up
                sup.show() ;
                ext.show() ;
                System.out.println(''sup.str = '' + sup.str) ;
                System.out.println(''ext.str = '' + ext.str) ;
        }
}
```

Résultat :

```
Extend.show : ExtendStr
Extend.show :ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

4.1.5 Super

super permet d'accéder au constructeur de la classe étendue. super doit être la **première instruction** du constructeur de la classe.

Si un constructeur de la classe étendue n'est pas explicitement appelé lors de la 1ère instruction, java l'appelle automatiquement. Si ce dernier n'existe pas on doit explicitement appeler un constructeur avec argument(s).

super permet d'accéder aux attributs et aux méthodes de la classe étendue.

4.1.6 Classes et méthodes abstraites

L'objectif d'une classe abstraite est de définir une partie de son implémentation et de laisser le reste à la charge des sous classes.

Une telle classe est déclarée :

```
abstract      class  NomClasse {
        // une méthode à surcharger
        abstract      void   uneMethodeASurcharger () ;

        // une méthode implémentée
        public        void   uneMethodeImplementee () {.....} ;
}
```

4.2 Package

Les packages permettent de regrouper des ensembles de classes. Ils sont l'équivalent des bibliothèques C.

Les utilitaires java sont livrés dans des packages.

Pour utiliser une classe d'un package 2 solutions :

1. Utiliser le nom complet de la classe :

```
java.util.Date.now = new java.util.Date() ;
```

2. Importer le package

```
import java.util.Date ;
Date.now = new Date() ;
```

Pour importer toutes les classes d'un package :

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

Le package java.lang est automatiquement importé. Il contient des classes comme Thread ou System.

Quelques packages java disponibles en version JDK 1.1

Nom du package	Contenu
java.applet	Classes de base pour les applets
java.awt	Interface Homme Machine
java.awt.datatransfert	Copier/Coller
java.awt.event	Evénements graphiques
java.beans	Composants java beans
java.io	Entrées/Sorties : fichier , pipe
java.lang	classes faisant partie du langague
java.math	utilitaires mathématiques
java.net	accès aux ressources du réseau
java.rmi	objets répartis
java.security	gestion de la sécurité
java.sql	accès aux bases de données
java.text	support pour l'internationalisation
java.util	conteneurs, date ...
java.util.zip	compression, décompression

Le compilateur utilise la variable d'environnement CLASSPATH pour localiser les fichiers contenant les classes sur le disque.

Exemple : CLASSPATH = ''c :\jdk1.1.6\lib\classes.zip ;c :\users\packages ;''

Les packages peuvent être rangés dans une arborescence ou dans un fichier compressé .zip.

Exercice : Héritage (Vehicule_7)

Créer une classe Autocar qui hérite de la classe Vehicule. Elle décrit en plus le nombre de sièges disponibles et le nombre de passagers. Créer le constructeur et les méthodes toString() et ModifierNombrePassagers(entier signé indiquant l'évolution du nombre de passagers) de la classe Autocar. La méthode ModifierNombrePassagers retourne vrai ou faux selon que le nombre de passagers est supérieur ou pas au nombre de sièges. Créer un Autocar, faire tourner l'autocar à gauche et modifier le nombre de passagers. Afficher ses caractéristiques.

Exercice : Classe abstraite (Vehicule_8)

Créer une classe abstraite Energie qui définit une méthode abstraite faireLePlein et une méthode getNiveau qui retourne le niveau courant. Cette méthode est étendue par deux classes Essense et Electricite. On ajoute dans la classe Vehicule un attribut Energie qui est une référence à un type d'énergie (Electricite ou Essence). Le type d'energie est fournie à la construction de l'objet. Ajouter une méthode demarrer à la classe Vehicule qui s'assure que le niveau n'est pas nul. Faire démarrer tous les véhicules et vérifier qu'ils font le plein selon la méthode qu'il convient.

5. Interfaces, Héritages multiple d'interfaces, conception par contrat

Les objectifs de ce chapitre sont :

1. Mise en œuvre des interfaces
2. Connaître l'héritage multiple des interfaces

5.1 Syntaxes

La syntaxe de déclaration est la suivante :

```
interface     NomDeLInterface [extends NomDeLInterfaceHeritee] {

              [déclaration des variables ;]
              [déclaration des méthodes ;]
}
```

La syntaxe d'utilisation de l'interface est

```
class  NomDeLaClasse implements NomDeLInterface {

              [déclaration des variables ;]
              [déclaration des méthodes ;]
}
```

Les méthodes déclarées dans une interface sont toujours public, jamais static.
Les variables sont toujours static et public.

Note : l'opérateur instanceof retourne true si l'objet implémente une interface :

```
if (lObjet instanceof lInterface) .....
```

Exemple :

```
interface     Affichage {
        void Afficher () ;
}
```

```
class Agent implements Affichage {
        private         String          _nom ;.
        private         Gare            _gareDAffectation ;

        void Agent (Gare GareDAffectation, String Nom) {...}

        void Afficher () {
                System.out.println (''Caractéristiques de l'agent\n'') ;
                System.out.println (''Nom : '' + _nom) ;
                System.out.println (‘Gare d'affectation : '' + _gareDAffectation ) ;
        }
}
```

5.2 Héritage multiple

Une classe peut hériter de plusieurs interfaces

class UneClasse implements UneInterface, UneAutreInterface, EncoreUneAutreInterface {....}

5.3 Variables d'interfaces

Ce sont forcément des variables : **static final**
De ce fait les mots clés static et final peuvent être omis.

Elles doivent être initialisées.

5.4 Utilisation

Dans le cadre de projets réalisés par plusieurs personnes les interfaces permettent de définir les obligations de chacun vis à vis des autres.

On définit ainsi les signatures des méthodes.

6. Classes de base

Les objectifs de ce chapitre sont :

1. Connaître la classe Object
2. Connaître les classes « Wrapper »
3. Connaître les classes liées aux chaînes de caractères
4. Manipuler les collections d'objets

6.1 Classe Object

La classe object est la classe dont héritent toutes les autres classes par défaut.

Ainsi toutes les classes héritent de ses méthodes.

public boolean equals (Object obj)

retourne vrai si l'objet en paramètre est égal à l'objet auquel s'applique la méthode. En fait cette méthode ne retourne vrai que si les 2 objets sont une même référence vers une même instance d'une même classe. Pour une comparaison attribut par attribut il faut surcharger cette méthode.

public final Class getClass ()

retourne l'objet Class associé à la classe de l'objet courant Cet objet décrit la classe de l'objet.

protected void finalize () throws Throwable

cette méthode est appelée avant que le garbage collector ne détruise un objet. Il est important, dans cette méthode de libérer les ressources systèmes (fichier, socket ...) avant que l'objet ne soit détruit. Pour optimiser les performances, le garbage collector ne libère pas systématiquement la mémoire utilisée par les objets. On peut forcer la libération par les méthodes System.gc() ou System.runFinalization().

6.2 Classes Wrapper

Les classes Wrapper sont des représentations sous forme de classes des types de base. Toutes les classes Wrapper offrent les méthodes suivantes :

un constructeur avec en paramètre le type de base.

```
Character    unCaractere      =      new   Character ('a') ;
```

un constructeur qui décode une String pour initialiser l'objet (sauf Character)

```
Integer      unEntier         =      new   Integer (''2456'') ;
```

une méthode toString qui transforme l'objet en une chaîne de caractères prête pour l'affichage.

```
Integer      unEntier ;
.....
System.out.println (''Valeur de l'entier = '' + unEntier.toString()) ;
```

une méthode qui transforme l'objet en son type de base

```
Boolean      unBoolean ;
boolean      unTypeDeBase ;
unTypeDeBase  = unBoolean .booleanValue() ;
```

une méthode equals pour comparer les valeurs de 2 objets Wrapper du même type.

Une méthode hashCode qui retourne une valeur utilisable dans les « hashtables »

Ci-dessous quelques méthodes spécifiques à chaque classe Wrapper.

Type	Classe Wrapper	Méthodes
boolean	Boolean	valueOf
int	Integer	public static int parseInt (String s) throws NumberFormatException ; public static Integer valueOf (String s, int radix) throws NumberFormatException ; ....
float	Float	public static float inBitsToFloat (int bits) ; ....
double	Double	public static double longBitsToDouble (long bits) ;
long	Long	public static Long valueOf (String str, int radix) ;
character	Character	public static int digit (char ch, int radix) ; public static char forDigit (int digit, int radix) ; public static boolean isLowerCase (char c) ; .... public static char toLowerCase (char ch) ; .....

Pour plus de renseignement consulter la documentation java. Celle ci est accessible sous forme de document, consultable depuis un navigateur à l'adresse :

file:///c :/jdk1.1.6/docs/api/packages.html (à vérifier en fonction de l'installation de java)

6.3 Classe java.lang.String

Cette classe permet de gérer les chaînes de caractères.

String UneChaine = new String (''Ceci est une chaîne de caractères'') ;

On peut créer un objet String implicitement sans utiliser l'opérateur new :

String UneChaine = ''Construction d'une chaîne de caractères sans new'' ;

Cette classe met à disposition des méthodes de traitement des chaînes de caractères :

Concaténation :

```
Integer     sexe ;
String      annee = ''98'' ;
String      mois = ''10'' ;
int         departement = 13 ;
String numSecuriteSociale = sexe.toString() + '' '' + annee ;
numSecuriteSociale += '' '' + mois ;
numSecuriteSociale += '' '' + departement; // La conversion du int en String est
rendu implicite par la présence du blanc.
```

Exercice : Classe Object (Vehicule_11)

Définir dans la classe Vehicule une méthode qui surcharge la méthode equals de la classe Object. La mettre en œuvre pour la comparaison de 2 véhicules et de 2 autocars.

Exercice : Classe String (String_1)

Ecrire une méthode pour convertir les nombres en nombres regroupés par 3 chiffres séparés par une virgule. Exemple 1234567890 devient 1,234,567,890.

- Longueur de la chaîne : uneChaine.length() ;

- Comparaison : uneChaine.equalsIgnoreCase (uneAutreChaine) ;

- Extraction d'une portion : portionExtraite = uneChaine.substring (indexDepart) ;

Attention : un objet String ne peut pas être modifié. A chaque concaténation il y a création d'un nouvel objet et destruction de l'ancien par la garbage collector dès qu'il n'est plus référencé.

6.4 Classe java.lang.StringBuffer

Cette classe représente une chaîne de caractères modifiable

Méthode	Définition
append (parametre)	ajoute en fin de chaîne la représentation sous forme d'une chaîne de caractères du paramètre passé (boolean, char, float, double, ... Object)
capacity()	retourne la capacité de la chaîne sans allocation complémentaire.
Insert (int offset, xxxx parametre)	insère à l'offset indiqué la représentation sous forme d'une chaîne de caractères du paramètre passé (boolean, char, float, double, ... Object)

En cas de dépassement de la capacité, le buffer est automatiquement redimensionné.

6.5 Classe java.util.Vector

C'est un tableau de références à taille variable. On ne peut pas y stocker directement des types de base. Il faut dans ce cas utiliser les Wrappers.

Cette classe fournit les méthodes suivantes :

- Insertion et suppression de références :

  addElement (Element)

  insertElementAt (Element, Position)

  removeElementAt (Position)

  removeElement (Element)

  removeAllElements ()

- Parcours du contenu

  indexOf (Element)

  contains (Element) ...

  elements() retourne une énumération

  la classe Enumeration a deux méthodes :

  boolean hasmoreElements () // retourne true tant qu'il y a des éléments

  Object nextElement() // retourne l'objet suivant de l'énumération.

Et bien d'autres méthodes.

Exercice : Classe Vector (Vehicule_10)

Utiliser dans le programme principal de l'exercice sur les Véhicules un vecteur pour définir un parc automobiles.

6.6 Classe java.util.Hashtable

C'est un tableau associatif permettant :

- d'indexer un objet par une clef
- de retrouver rapidement un objet à partir d'une clef

Les clefs sont des objets d'une classe qui doit implémenter les méthodes :

- equals : pour pouvoir comparer deux clefs et,
- hashCode (méthode de la classe Object) : pour identifier de manière unique l'instance d'une classe parmi toutes les instances d'une classe.

Dans la pratique, la clef est une String, classe pour laquelle ces 2 méthodes sont redéfinies.
Cette classe fournit les méthodes suivantes :

Object put (Object key, Object value) : // insère un élément.
Object remove (Object key) : // supprime la clef et l'objet associé de la table
Object get (Object key) : // retourne la valeur associée à la clef
boolean containsKey (Object key) // teste si la clef existe
Enumeration keys() // retourne une énumération sur les clefs
Enumeration elements() // retourne une énumération sur les valeurs.

7. Les exceptions

Les objectifs de ce chapitre sont :

1. Comprendre le besoin
2. Savoir créer et utiliser les exceptions
3. Déclencher une exception
4. Connaître la classification des exceptions
5. Utiliser le polymorphisme avec les exceptions

7.1 Principe

Gérer toutes les situations « anormales » de façon indépendante du traitement normal.

Le but étant d'alléger le traitement des erreurs. Gérer toutes les erreurs possibles devient souvent inextricable.

Un compte-rendu est utilisé pour traiter des cas précis,

Une exception est utilisée pour traiter tous les autres cas : débordement d'index, erreur d'entrée/sortie ...

Exemple

```
class PasDeSolution extends Exception {}

class EquationSecondDegre {

        public double solution1 () throws PasDeSolution {
                .....
                if (discriminent < 0) throw new PasDeSolution
        }
        public double solution2 () throws PasDeSolution {
                .....
                if (discriminent < 0) throw new PasDeSolution
        }
}


class Test {
        EquationSecondDegre    equa = new EquationSecondDegre(...) ;
        double                 resultat ;
        try {
                resultat = equa.solution1() ;
                ....
                System.out.println (''résultat = '' + resultat) ;
        } catch (PasDeSolution e) {
                System.out.println (''pas de solution'') ;
        }
}
```

7.2 Spécification de l'exception : throws

Une méthode doit déclarer les exceptions qu'elle est susceptible de lever.

public solution1 (.....) **throws** PasDeSolution

Ainsi l'utilisateur de cette méthode est averti, sans entrer dans le code, que cette méthode risque de lever une exception.

Le compilateur vérifie que toutes les exceptions sont traitées.

7.3 Utilisation de l'exception : try, catch, finally

Toute utilisation d'une méthode pouvant lever une exception doit être incluse dans un bloc : **try, catch**

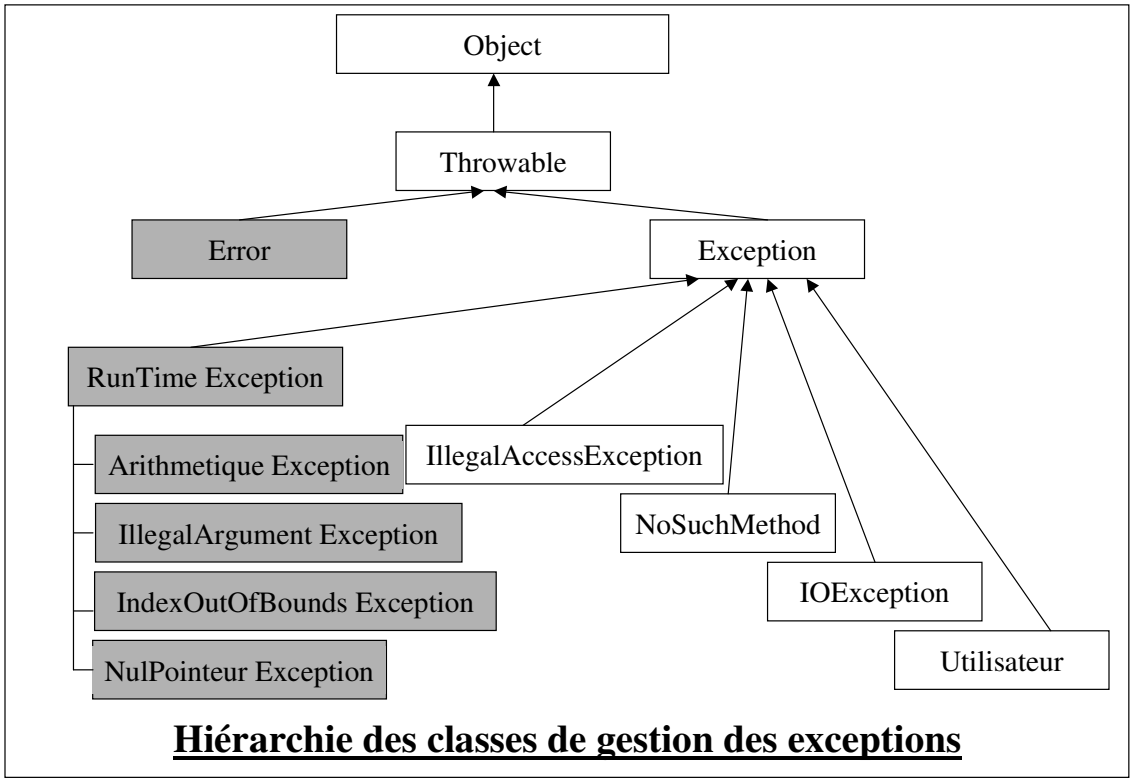
```
try {
        resultat = equa.solution1() ;
        ....
} catch (PasDeSolution e) {
        .....
} finally {
        // libérer les ressources
}
```

Dès que la méthode « solution1() » remonte l'exception, le traitement est dérouté vers le bloc catch de l'appelant.

Qu'il y ait retour ou pas d'une exception le bloc **finally** est toujours exécuté.

Le bloc finally est toujours exécuté même après une instruction **return** dans un bloc

7.4 Classification des exceptions



Hiérarchie des classes de gestion des exceptions

Les exceptions sont hiérarchisées.

Les exceptions de type Error, RunTimeException non pas besoins d'être déclarées, le compilateur n'impose pas leur traitement. Toutes les autres doivent impérativement être traitées.

7.5 Exceptions et polymorphisme

Il est conseillé de concevoir un système d'exception métier hiérarchique.

Exemple : gestion d'une connexion à un serveur :

```
classe ExceptionDeConnexion
        classe ExceptionDefautReseau
        classe ExceptionServeurAbsent
        classe ExceptionIdentification
                classe ExceptionMotDePasse
                classe ExceptionUtilisateurInconnu

try {
        UnServeur.seConnecter () ;
} catch (ExceptionServeurAbsent) {
        // Serveur absent
} catch (ExceptionIdentification) {
        // Identification en echec
}
```

La machine exécute le **premier bloc compatible** de l'exception levée et n'en exécute **qu'un seul**. De ce fait « catcher » une super exception avant une sous exception est une erreur de compilation.

7.6 Debugging

La classe exception dérive de la classe Object. La méthode toString() est disponible pour identifier l'exception levée.

PrintStackTrace() affiche la pile des appels depuis la méthode qui a déclenché l'exception.

La classe Throwable, accepte une String à sa construction. La méthode getMessage() restitue ce message.

7.7 Propagation des exceptions

Lorsque la méthode appelante reçoit une exception elle peut, si elle ne sait pas la traiter la propager vers son propre appelant.

```
catch (Throwable e) {
        System.out.println (''SOS une exception de type : '' + e.toString()) ;
        throw e ;
}
```

Exercice : Exceptions (Vehicule_9)

Dans la classe Autocar, lever une exception de type SiegeException lorsque l'on tente de définir un nombre de sièges négatif à la création. Définir une méthode toString dans la classe SiegeException. Créer un autocar avec un nombre négatif de sièges et récupérer l'exception pour afficher un message d'erreur incluant la méthode toString de l'exception levée.

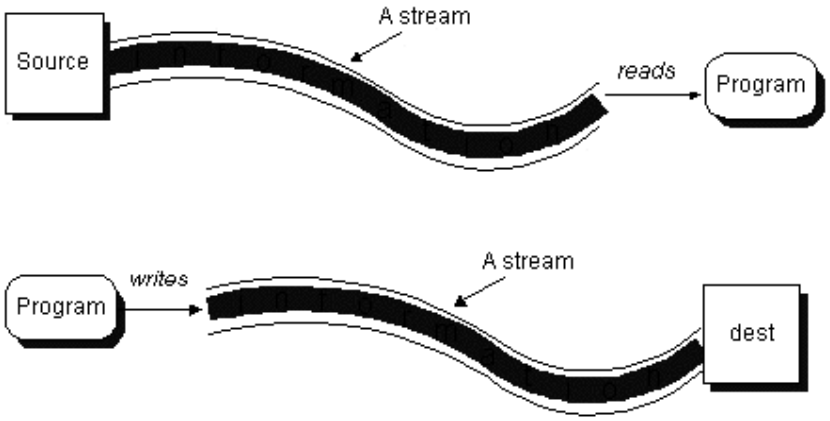
8. Entrées / Sorties

Les objectifs de ce chapitre sont :

1. Connaître les classes de base
2. Savoir lire et écrire un flot de données binaires
3. Savoir lire et écrire un flot de données texte

8.1 Principes

Attention : Les applets doivent satisfaire aux contraintes de sécurité. L'accès au disque du client est à priori interdit



Un canal (source, destination) est une source ou un puits de données : fichier, pipe, socket...

Un flux (stream) pour se connecte à un canal : il offre les méthodes de lecture et d'écriture de haut niveau quelque soit le canal.

L'algorithme est toujours le même :

1. Ouvrir le flux,
2. Lire ou écrire tant qu'il y a des informations
3. Fermer le flux.

8.2 Structure des classes

On peut organiser les classes selon deux critères :

Le type de donnée qu'elles traitent

- Octets
- Caractères

Le type de fonctionnalité qu'elles offrent.

- Accès aux données
- Traitement des données.

8.3 Les accès aux données

Type de source ou de destination	Flux de caractères	Flux d'octets
Mémoire (tableau de caractères ou d'octets)	`CharArrayReader`, `CharArrayWriter`	`ByteArrayInputStream`, `ByteArrayOutputStream`
Mémoire (une chaîne de caractères « String »)	`StringReader`, `StringWriter`	
Pipe (flux de données en mémoire via un buffer circulaire)	`PipedReader`, `PipedWriter`	`PipedInputStream`, `PipedOutputStream`
Fichier	`FileReader`, `FileWriter`	`FileInputStream`, `FileOutputStream`

8.4 Le traitement des données

Traitement	Flux de caractères	Flux d'octets
Stockage pour lecture par paquets : public int **read**(char[] cbuf, int off, int len) throws IOException public void **write**(char[] cbuf, int off, int len) throws IOException public String **readLine**() throws IOException public void **write**(String s, int off, int len) throws IOException	`BufferedReader`, `BufferedWriter`	`BufferedInputStream`, `BufferedOutputStream`
Sélecteur (classes destinées à être étendues par d'autres)	`FilterReader`, `FilterWriter`	`FilterInputStream`, `FilterOutputStream`
Conversion entre octets et caractères public **InputStreamReader**(InputStream in, String enc) throws UnsupportedEncodingException	`InputStreamReader`, `OutputStreamWriter`	
Concaténation de flux public **SequenceInputStream**(Enumeration e) (énumération d'InputStream)		`SequenceInputStream`
Sérialisation d'objets		`ObjectInputStream`, `ObjectOutputStream`

Traitement	Flux de caractères	Flux d'octets
Conversion de données (codage des types de base Java indépendant de la machine) public final double **readDouble**() throws IOException public final int **readInt**() throws IOException		`DataInputStream`, `DataOutputStream`
Comptage : conserve le nombre de lignes lues `public int` **`getLineNumber`**`()`	`LineNumberReader`	`LineNumberInputStream`
« Peaking ahead » flux autorisant les relectures public void **unread**(byte[] b, int off, int len) throws IOException	`PushbackReader`	`PushbackInputStream`
Impression de texte (et non pas de graphique) public void **print**(String s)	`PrintWriter`	`PrintStream`

Exemple

```
import java.io.*;

class EntreeSortie {

  // le String argv [] est indispensable
  public static void main (String argv[])
  {
    InputStream in = System.in;
    OutputStream out = System.out;

    System.out.println ("\n\nDébut de l'exemple d'entrée, sortie, saisissez une
ligne de caractères : ");
    try {
      for (int i = 0; i < 5; i++) {
        // Lecture du clavier
        int x = in.read ();
        // Affichage après transformation
        out.write (Character.toUpperCase ((char) x));
      }
    } catch (IOException e) {
      System.out.println ("Exception : " + e.toString());
    }

    System.out.println ("\n\nFin de l'exemple d'entrée, sortie");
  }

}
```

8.5 Classe java.io.File

Elle permet de gérer des fichiers.

Les constructeurs :

- public File (File dir, String name) ;
- public File (String path) ;
- public File (String path, String name) ;

Les méthodes disponibles sont :

- canRead(), canWrite() : peut on lire ou écrire dans le fichier ou le répertoire.
- delete() : supprime le fichier
- exists() : teste si le fichier existe.
- ....

Exercice : Entrées/Sorties (EntreeSortie)

Ecrire une classe qui lit le contenu d'un fichier et le recopie dans un autre fichier après transformation de tous les caractères de minuscule en majuscule.

Exercice : Entrées/Sorties, Hashtable, Exceptions

Gérer le parc auto avec la classe Hashtable. Ecrire une méthode rechercherVehicule qui prend au clavier le nom du propriétaire et affiche les caractéristiques de son véhicule.

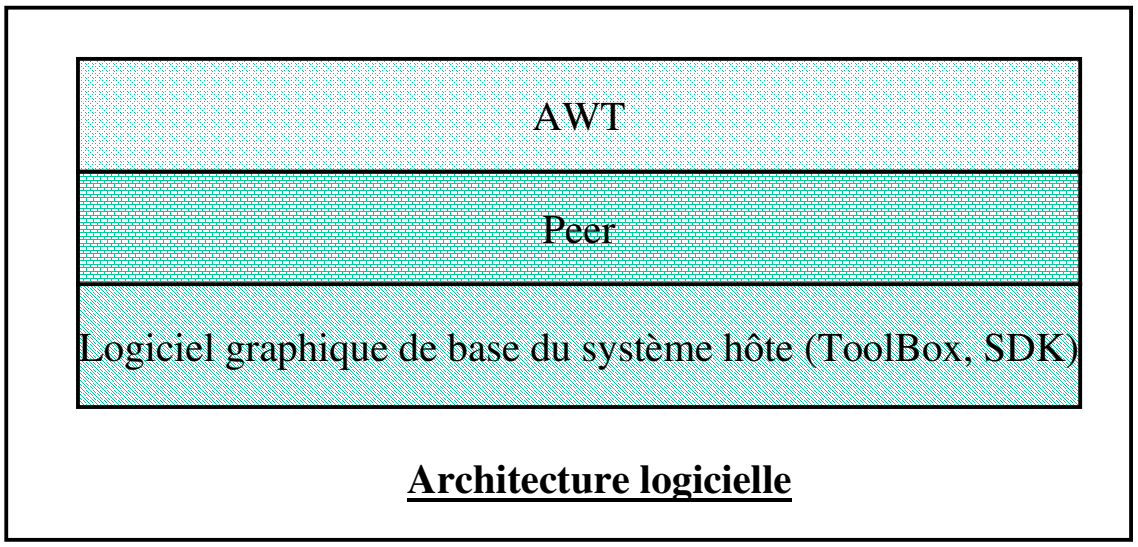
9. IHM

Les objectifs de ce chapitre sont :

1. Connaître AWT et les Swings (depuis JDK 1.2).
2. Assimiler le modèle Component/Container
3. Connaître les techniques de placement des composants (LayoutManager)
4. Assimiler le modèle d'événements du JDK

9.1 Architecture logicielle

N'oublions pas que java doit être portable. Il doit donc être indépendant de la plate-forme d'exécution.

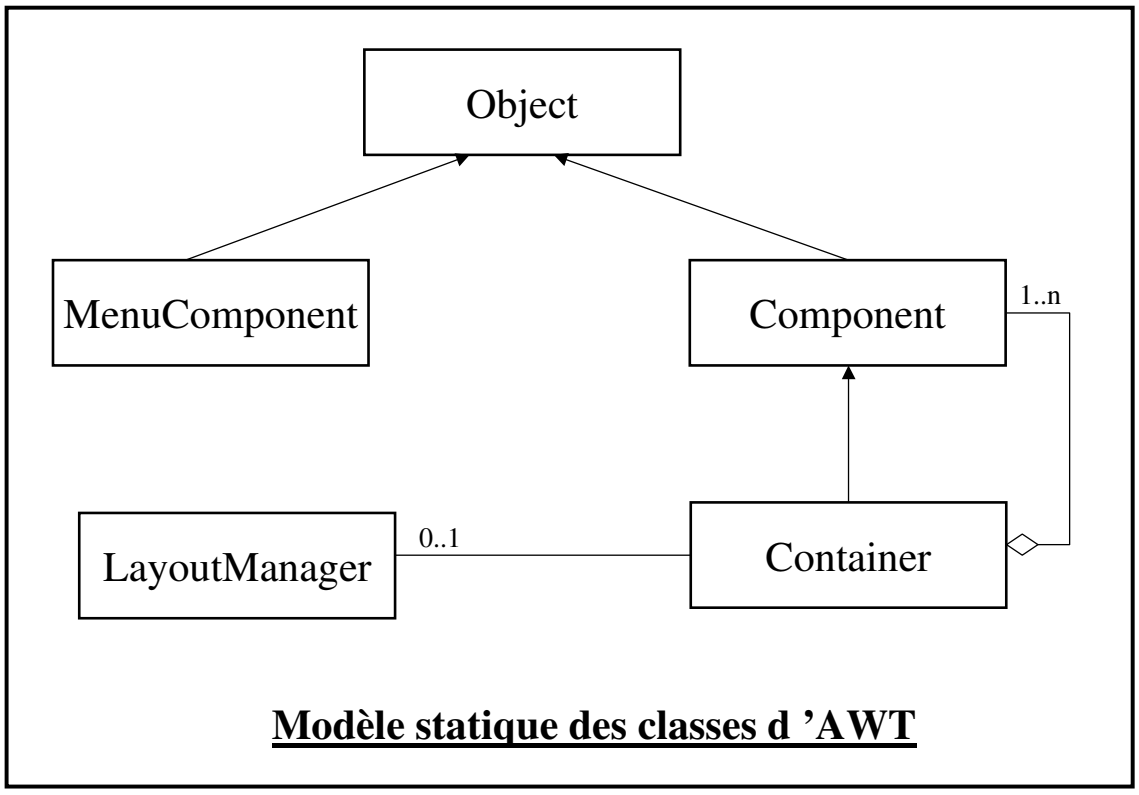


Architecture logicielle

Java utilise donc une couche « d'abstraction » qui permet de gérer toutes les plate-formes de la même façon : Abstract Window Toolkit.

La « Peer » réalise l'interface entre cette couche « d'abstraction » et la couche graphique native.

9.2 Modèle statique d'AWT



MenuComponent : C'est la super classe des classes MenuBar et MenuItem.

Component : C'est la super classe de tous les objets graphiques : bouton, ascenseur, texte, radio bouton ....

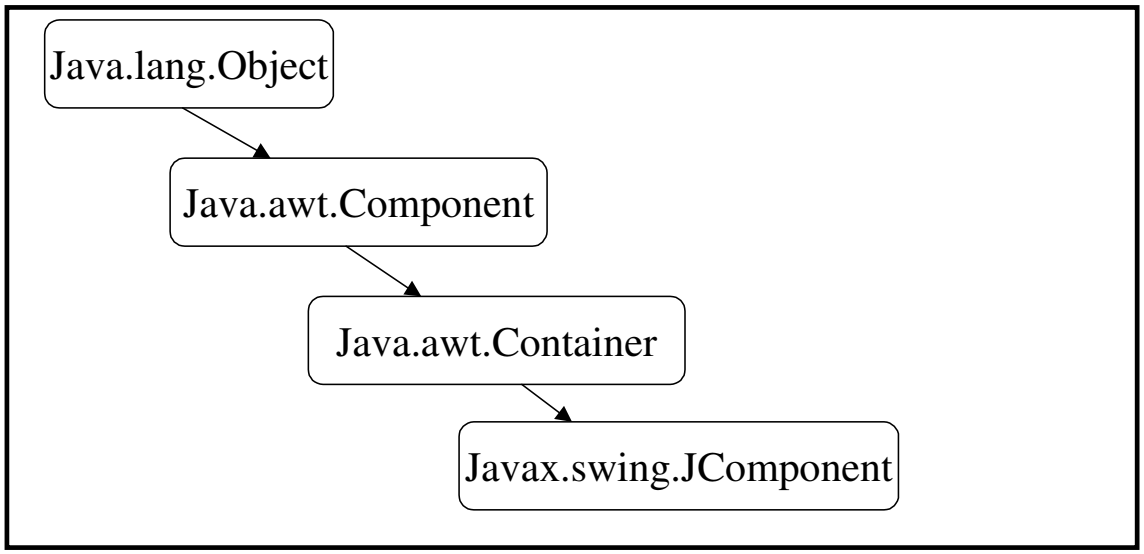
Container : un « container » est un « component ». Un « container » peut contenir un ou plusieurs « component » : fenêtre, « panel » (espace graphique pour disposer des composants).

LayoutManager : organise la disposition des composants graphiques et ceci quelque soit la taille de la fenêtre.

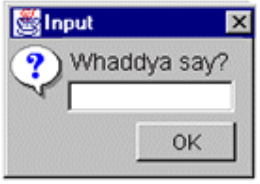
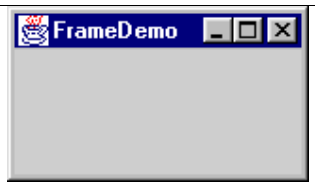
9.3 Modèle statique des Swing

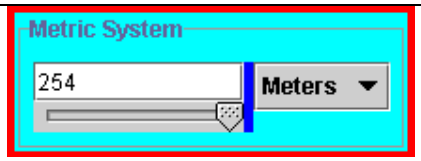
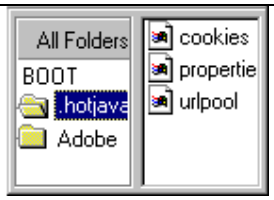
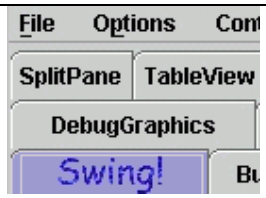
Les Swing n'utilisent pas directement le code natif de la plate-forme. Leur aspect est donc indépendant de la plate-forme.

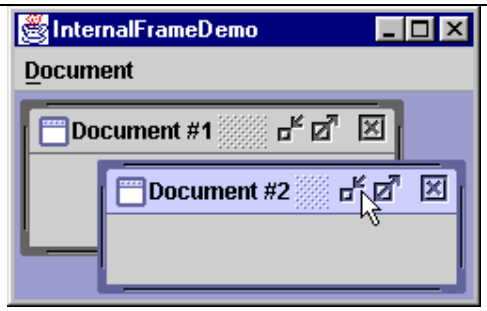
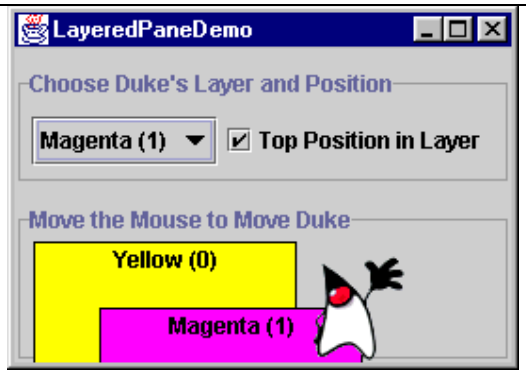
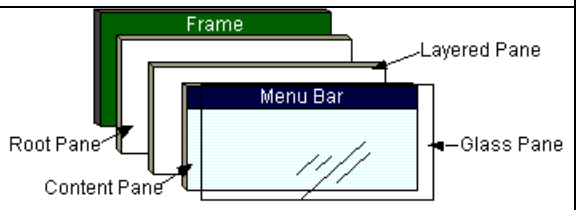
Ils offrent des fonctionnalités supplémentaires
- Bouton incluant des images ou de forme non rectangulaire
- Objets plus évolués : JTable, JTree.

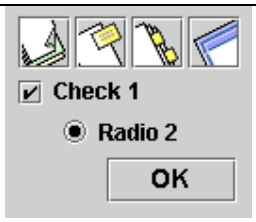
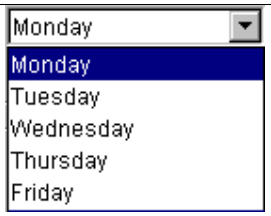
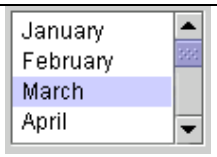
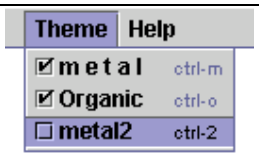
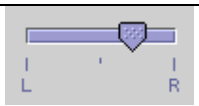


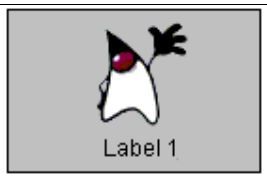
9.4 Les conteneurs

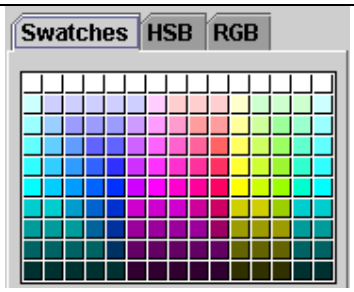
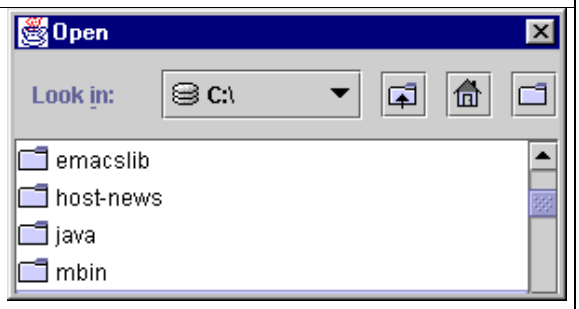
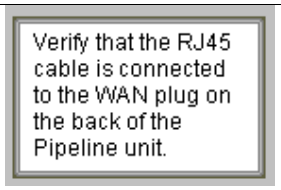
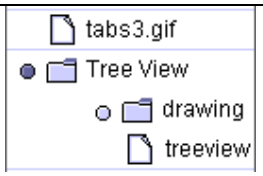
Conteneurs du plus haut niveau	
JOptionPane : Fenêtre de dialogue utilisée pour informer, demander une confirmation ou une saisie d'information.	
JFrame : Fenêtre	

Conteneurs généraux	
JPanel : Conteneur	
JScrollPane : Conteneur avec les ascenseurs	
JSplitPane : Conteneur divisé en 2 sous conteneurs (horizontalement ou verticalement).	
JTabbedPane : Conteneur présentant un composant parmi N (onglets)	
JToolBar : Conteneur de plusieurs composants, généralement des boutons alignés horizontalement ou verticalement	

Conteneurs spécifiques	
JInternalFrame : Une fenêtre disposée à l'intérieur d'une autre fenêtre.	
JLayeredPane : Un conteneur avec la notion de profondeur. Chaque composant est ajouté à un niveau donné. On peut activer la visualisation d'un niveau parmi N.	
JRootPane : le conteneur de base. En général on ne crée pas directement un JRootPane. Il est automatiquement créé lorsqu'on crée un conteneur du plus haut niveau (JInternalFrame, JApplet, JDialog et JFrame).	

Objets de base	
JButton, JRadioButton, JCheckBox	
JComboBox	
JList	
JMenu	
JSlider	
JTextFiel	

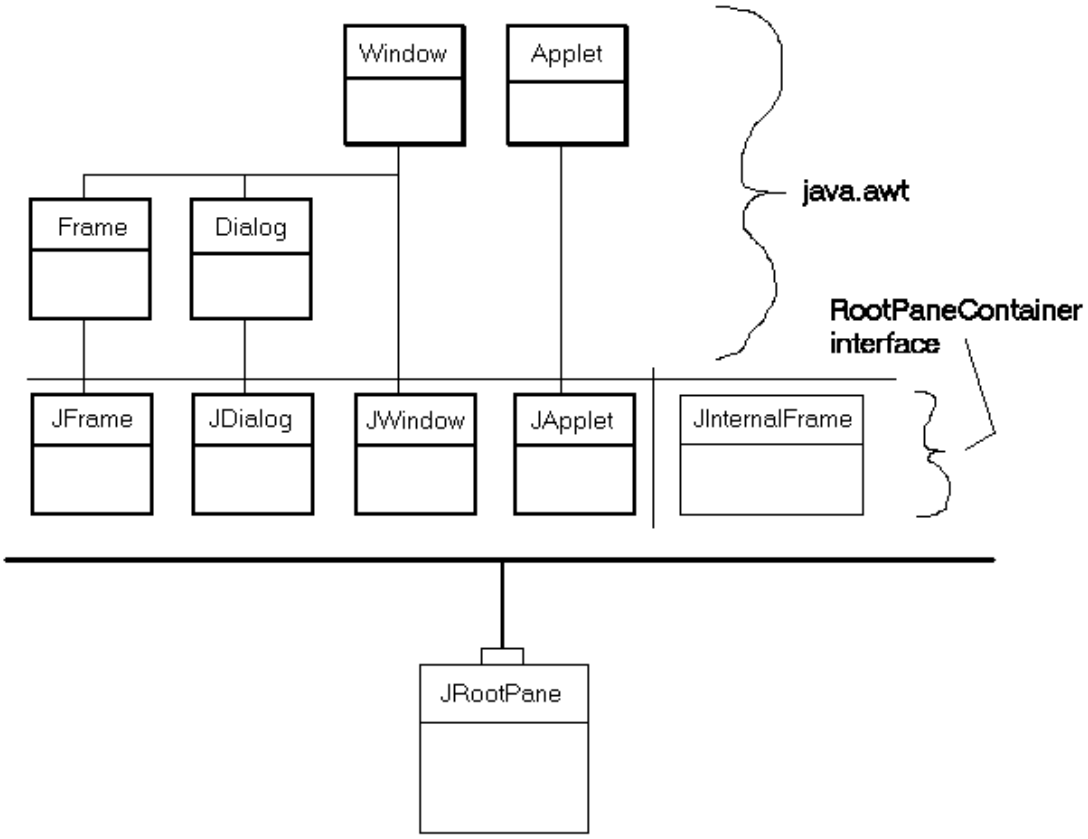
Objets non modifiables par l'opérateur	
JLabel	
JProgressBar	
JToolTip : Texte associé à un composant, visualisé lorsque ce dernier est désigné par le curseur.	

Objets plus complexes modifiables par l'opérateur	
JColorChooser	
JFileChooser	
JTable	
JTextArea	
JTree	

9.5 Les containers

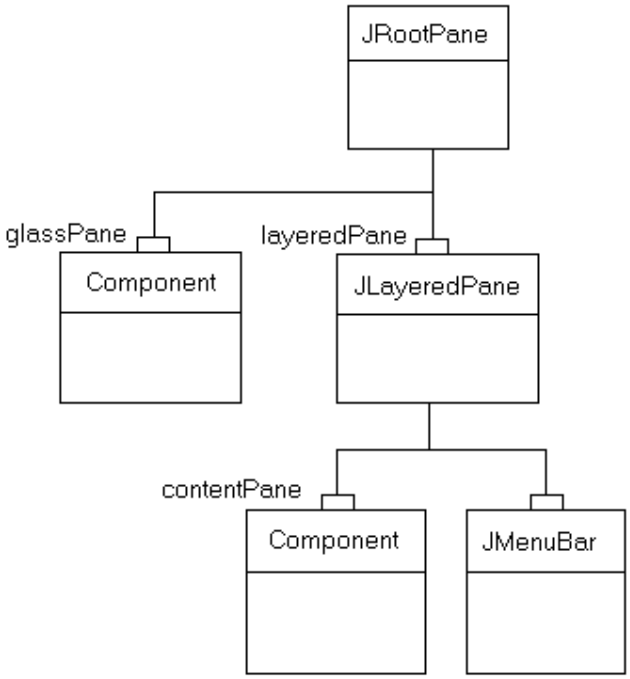
Un « container » est un composant pouvant inclure d'autres composants.

Les containers :



Le JRootPane définit la structure de base des 5 conteneurs JFrame, JDialog, JWindow, JApplet et JInternalFrame.

9.6 Le JRootPane



Le JRootPane est constitué d'un « glassPane » (conteneur du plus haut niveau dédié à la récupération des événements), d'une barre de menu (optionnelle) et d'un « contentPane » (conteneur). Les positions de la barre de menu et du « contentPane » sont gérés par le « layeredPane »

9.7 Les « LayoutManager »

Un layout manager est associé à un container. Il assure le positionnement des composants insérés dans le container.

Par défaut aucun layout manager n'est associé à un container. Dans ce cas c'est à l'application de positionner les composants. Il est possible de développer son propre layout manager.

Les différents types de layout manager disponibles sont :

- FlowLayout,
- BorderLayout,
- GridLayout,
- BoxLayout,
- OverlayLayout
- GridBagLayout,

9.7.1 FlowLayout

Le flowLayout dispose les composants les uns à la suite de l'autre. Quand il n'y a plus de place sur la ligne le composant est placé à la ligne suivante.

9.7.2 BorderLayout

C'est le layout manager par défaut d'une Frame (fenêtre avec barre de défilement, bouton de fermeture ...). Ce type de layout manager divise la frame en 5 régions : North, South, East, West, Center. On spécifie dans quelle partie du container on place le composant au moment de l'ajout.

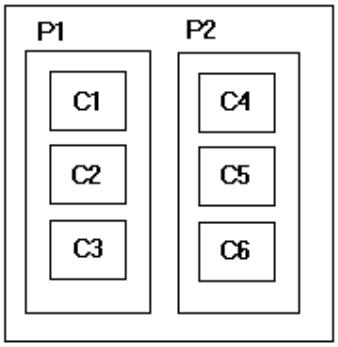
Dans les régions North et South les composants sont dilatés horizontalement pour occuper tout l'espace.
Dans les régions East et West les composants sont dilatés verticalement pour occuper tout l'espace.
Dans la région Center les composants sont dilatés horizontalement et verticalement pour occuper tout l'espace.

9.7.3 GridLayout

Le GridLayout dispose les composants dans une grille où chaque élément est de la même taille.

9.7.4 BoxLayout

Le BoxLayout dispose les composants verticalement ou horizontalement sans « retour à la ligne » quelque soit la taille du conteneur.



9.7.5 OverlayLayout

Le OverlayLayout dispose les composants les uns au dessus des autres.

9.7.6 GridBagLayout

Le GridBagLayout est une grille virtuelle, où chaque ligne et chaque colonne est de la taille du plus grand composant.

Son utilisation met en œuvre la classe GridBagConstraints (Cf. §7.5.6)

9.7.7 GridBagConstraints

La classe GridBagConstraints permet de positionner les composants lorsque l'on utilise un GridBagLayout.

Les attributs de cette classe sont :

inset : marges d'un composant dans sa zone d'affichage : bottom, left, right, top.

gridx, gridy : position dans la grille : gridx = 0 à gauche, gridy = 0 en haut.

gridwith, gridheight : nombre de cases horizontales et verticales de la zone d'affichage occupée.

fill : indique comment le layout manager doit retailler le composant pour remplir sa zone (BOTH, HORIZONTAL, VERTICAL, NONE).

weightx, weighty : indique comment distribuer l'espace disponible entre chaque cellule de la grille.

anchor : indique comment positionner dans sa zone un composant plus petit qu'elle : (EAST, WEST, NORTH, CENTER, NORTHEAST ...)

ipadx, ipady : marge du composant dans sa zone.

9.8 Utilitaires

Java.awt.color :

- Permet de définir ses propres couleurs selon le modèle RGB et 256 niveaux.
- Couleurs prédéfinies (color.blue, color.cyan, color.black ...)
- Méthodes de conversion d'un modèle vers un autre (RGBtoHSB)

java.awt.Font et java.awt.FontMetrics

- Gestion, création d'une fonte de caractères.

9.9 Java.awt.Graphics

C'est la classe de base qui définit le contexte graphique de dessin.

Un contexte graphique est un attribut d'un composant. Avant de l'utiliser il faut récupérer une référence sur cet objet : component.getGraphics()

On dispose en suite de méthodes permettant de :

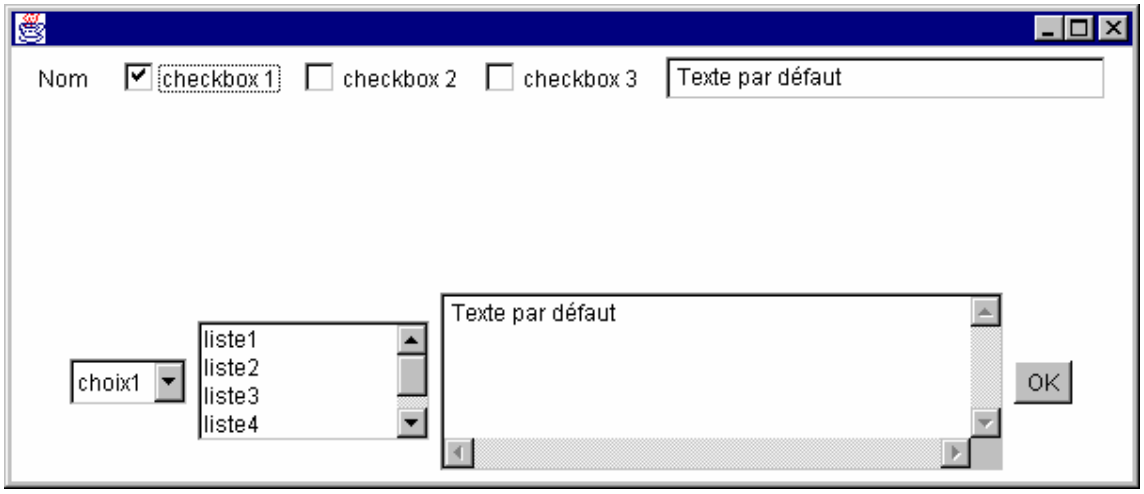
- définir un système de coordonnées logiques,
- une zone de clipping,
- la couleur de tracé,
- la fonte,
- de réaliser des tracés : lignes, rectangles pleins ou vides, des cercles ...
- le mode de tracé (normal, XOR ...).

Exercice : GridLayout (awt)

Créer une frame dans laquelle on dispose selon une grille de 10 lignes et 10 colonnes 100 boutons. Chaque bouton aura une couleur différente (dégradé).

Exercice : Composants Swing de base (Swing_1)

Ecrire une classe « Fenetre » (JFrame) qui permet d'afficher dans 2 panels un en haut l'autre en bas de la fenêtre les composants tels qu'ils sont dans la présentation des différents composants. Un objet de la classe « Fenetre » est créé depuis le programme principal.



9.10 Screen Updater

Pour dessiner ou rafraîchir un composant, on utilise la méthode repaint(). Cette méthode activera dès que possible le Screen Updater.

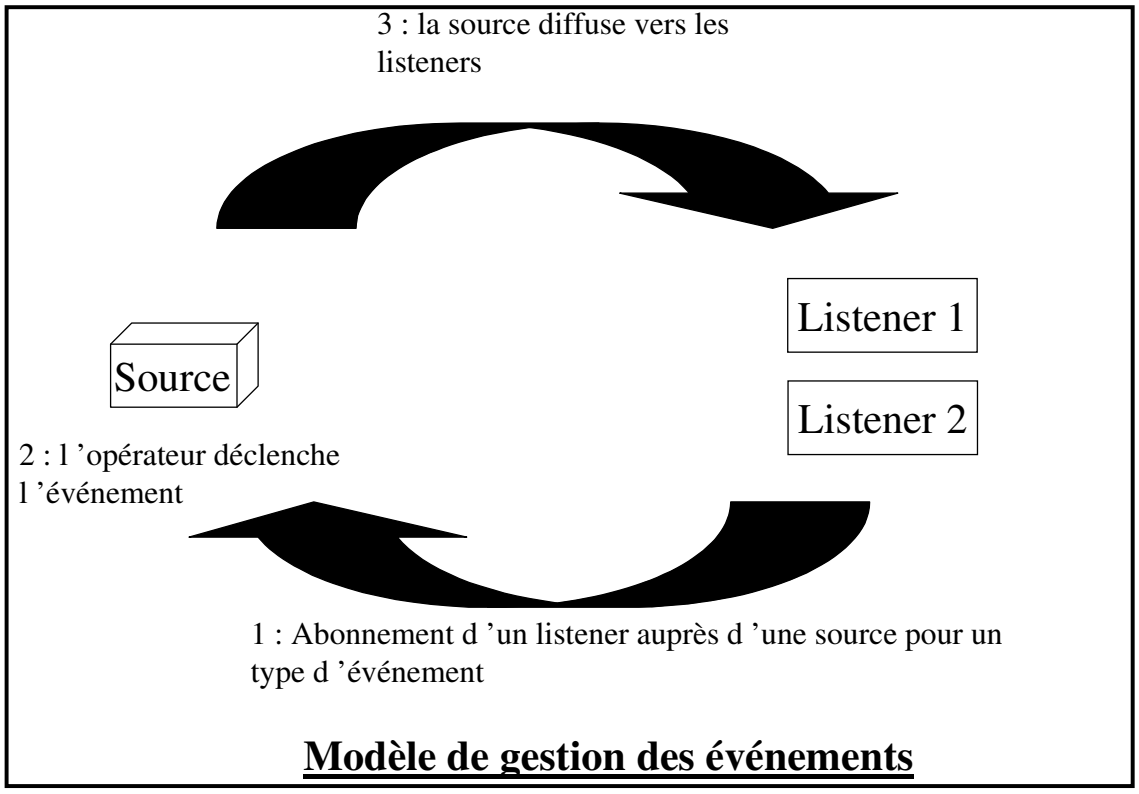
Le Thread Screen Updater lance la méthode Component.update (Graphic g)

La méthode Component.update(Graphic g) efface le composant avec la couleur de fond et active la méthode Component.paint().

9.11 Les événements

9.11.1 Principe sources, listeners

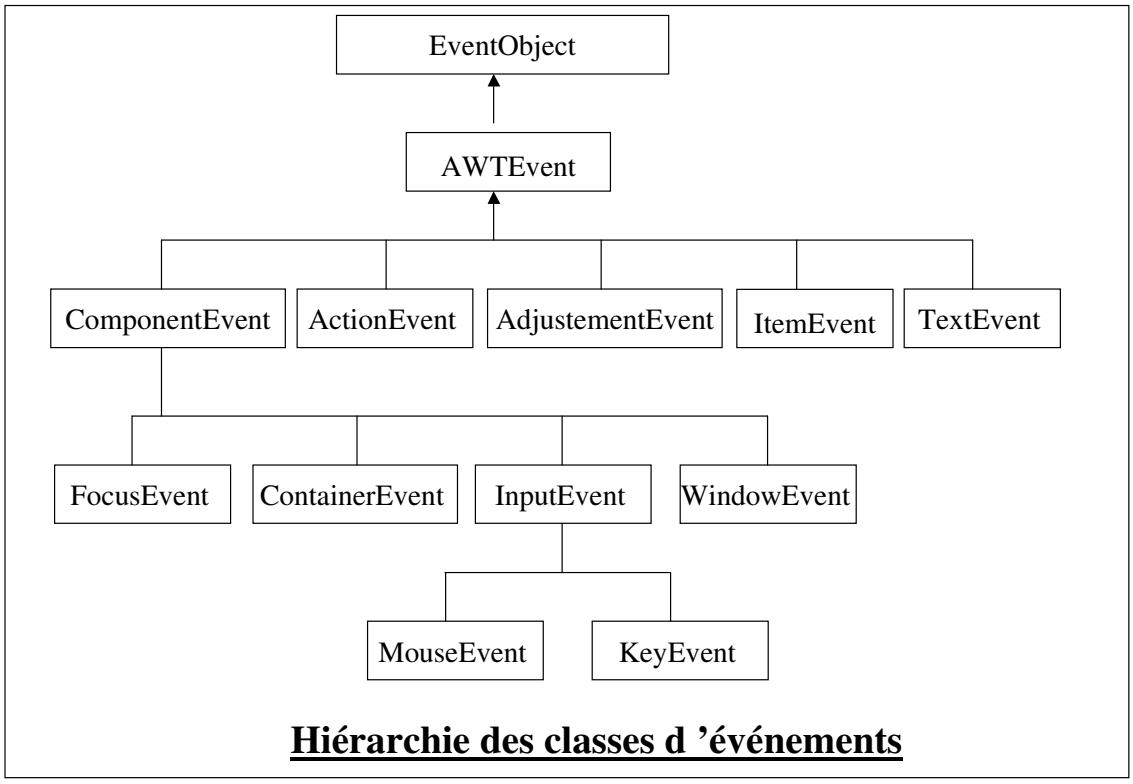
Le traitement des événements fonctionne sur le principe des méthodes postées. C'est à dire qu'un gestionnaire d'événements signale à une source d'événements qu'il est intéressé par le type d'événement X et que sur ce type d'événement X on doit activer sa méthode Y.



Modèle de gestion des événements

- N'importe quel Composant est une source potentielle.
- Les événements sont délivrés de façon synchrône.
- Plusieurs listeners peuvent s'abonner à une même source.

9.11.2 Les événements



Hiérarchie des classes d 'événements

9.11.3 Les Listeners

Tout objet qui s'abonne à une source d'événements « implements » une interface.

Il existe un type d'interface par classe d'événement :

- ActionListener,
- MouseListener,
- WindowListener,
- KeyListener ...

Chaque interface définit les méthodes à implémenter. MouseListener définit en autre :

- mouseClicked : invoquer lorsque l'on clique sur le composant,
- mouseEntered : invoquer lorsque le curseur entre dans le composant,
- mouseExited : invoquer lorsque le curseur quitte le composant,

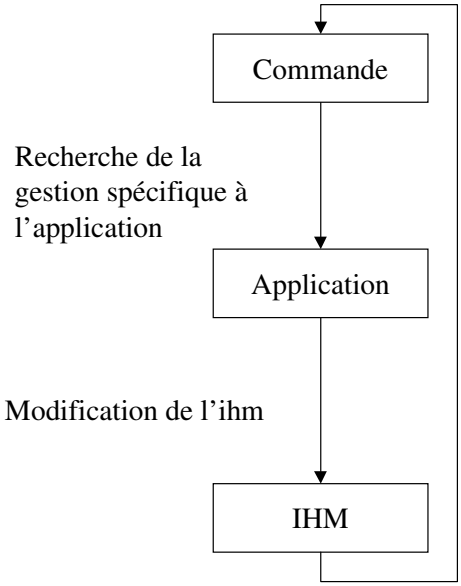
9.11.4 Modèle de conception objet adapté aux IHM Java

Le modèle source, listener de java influe sur le découpage d'une application java disposant d'une interface graphique.

Une application est découpée en 3 sous ensembles :

1. l'IHM proprement dite = ensemble des classes visualisant les objets graphiques
2. le gestionnaire des événements = ensemble des classes gérant les actions opérateur,
3. les objets métiers = ensemble des classes propres aux objets métier manipulés.

On découple ainsi fortement la partie graphique de la partie métier.



Modèle conceptuel des listeners

9.11.5 Exemple

```
public class IHM_2 {

  public static void main (String args[])
  {
     Application_2 application = new Application_2 ();
     Fenetre_2     fenetre = new Fenetre_2 (application);
  }
}

public class Application_2 {

  public Application_2 ()
  {
     System.out.println ("Création de l'application 2");
  }
  public   void   Valider () {
    System.out.println ("Fonction Valider");
  }
  public   void   Abandonner () {
    System.out.println ("Fonction Abandonner");
  }
  public   void   Terminer () {
    System.out.println ("Fonction Terminer");
    System.exit(0);
  }
}
```

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


class Commandes_2 implements ActionListener {
  static final  int     BOUTON_VALIDER = 0;
  static final  int     BOUTON_ABANDONNER = 1;
  static final  int     BOUTON_TERMINER = 2;

  private       int             _identite;
  private       Application_2   _application;
  private       Fenetre_2       _fenetre;

  public Commandes_2 (int in_Identite, Application_2 in_Application, Fenetre_2 in_Fenetre)
{
    _identite = in_Identite;
    _application = in_Application;
    _fenetre = in_Fenetre;
  }

  public void actionPerformed (ActionEvent in_e) {
    switch (_identite) {
    case BOUTON_VALIDER :
      _application.Valider ();
      break;
    case BOUTON_ABANDONNER :
      _application.Abandonner ();
      break;
    case BOUTON_TERMINER :
      _application.Terminer ();
      break;
    default :
      System.out.println ("Erreur : objet Commande inconnu, identite = " + _identite);
    }
  }
}
```

```
import java.awt.*;

class Fenetre_2  extends Frame {

  Panel    _panel;
  Commandes_2 _valider;
  Commandes_2 _abandonner;
  Commandes_2 _terminer;
  Button   _boutonValider;
  Button   _boutonAbandonner;
  Button   _boutonTerminer;

  Fenetre_2 (Application_2 in_application) {

    _panel = new Panel ();
    add (_panel, "South") ;

    // Création des boutons
    _boutonValider = new Button ("Valider");
    _panel.add (_boutonValider);
    _boutonAbandonner = new Button ("Abandonner");
    _panel.add (_boutonAbandonner);
    _boutonTerminer = new Button ("Terminer");
    _panel.add (_boutonTerminer);

    // Création des listeners
    _valider = new Commandes_2 (Commandes_2.BOUTON_VALIDER,
                        in_application,
                        this);
    _abandonner = new Commandes_2 (Commandes_2.BOUTON_ABANDONNER,
                          in_application,
                          this);
    _terminer = new Commandes_2 (Commandes_2.BOUTON_TERMINER,
                         in_application,
                         this);

    // Connexion des boutons aux actions
    _boutonValider.addActionListener (_valider);
    _boutonAbandonner.addActionListener (_abandonner);
    _boutonTerminer.addActionListener (_terminer);

    pack ();
    setVisible (true);
  }
}
```

Exercice : Listeners (IHM_2, Application_2, Fenetre_2)

Créer une fenêtre (JFrame) dans laquelle on présente une zone de saisie multi lignes (JTextArea) ainsi que 2 boutons (JButton) Valider, et Abandonner.
L'opérateur peut saisir un texte puis :

- Abandonner : tout le texte est effacé, la saisie peut recommencer,
- Valider : tout le texte saisi est affiché dans les traces de l'application. L'application se termine.

10. Impressions

L'objectif de ce chapitre est la gestion des impressions

10.1 Le package java.awt.print

Il est constitué des classes suivantes :

- La classe Book représente un document composé de plusieurs pages chaque page ayant son propre format.
- La classe PageFormat décrit la taille et le format de chaque page.
- La classe Paper décrit une feuille de papier.
- La classe PrinterJob est la classe de contrôle des impressions.

10.2 La classe PrinterJob

Elle fournit les services :

- getPrinterJob() : restitue une référence sur l'objet PrinterJob de gestion des accès aux impressions.
- setPrintable(Printable, PageFormat) : indique quel est l'objet à imprimer et selon quel format.
- printDialog() : présente à l'opérateur une fenêtre de dialogue pour modifier les caractéristiques d'impression.
- print() : lance l'impression de l'objet à imprimer.

10.3 L'interface Printable

L'objet à imprimer doit implémenter l'interface Printable. Cette interface définit une seule méthode :

public int **print**(Graphics graphics, PageFormat pageFormat, int pageIndex) throws PrinterException

où

- graphics est le contexte graphique de l'imprimante dans lequel il faut dessiner.
- pageFormat est le format de la page en cours
- pageIndex est l'index de la page (débute à 0).

Pour reproduire dans le contexte graphique de l'imprimante les composants présents dans la fenêtre à imprimer on active la méthode print() de l'objet de plus haut niveau qui se charge d'activer les méthodes print() des objets inclus.

Attention en cas de surcharge de la méthode print() il ne faut pas oublier d'activer la méthode surchargée pour ne pas rompre la chaîne.

Exercice : Impression (PrintMessage)

Créer une fenêtre (JFrame) dans laquelle on présente une zone de saisie (JTextField) 1 bouton (JButton) Imprimer et un texte (JLabel). L'opérateur peut saisir un texte puis imprimer la fenêtre.