Introduction

La qualité d'un logiciel se mesure par

- La qualité du produit : fonctionnalité, fiabilité, convivialité, conformité, …
- La qualité du processus de développement : maintenabilité, modularité, réutilisabilité, portabilité, …

Répartition de l'effort

- Spécification (6%)
- Conception (5%)
- Codage (7%)
- Tests-Validation (15%)
- Maintenance (67%)

Origine des erreurs de développements

- Spécification (56%)
- Conception (27%)
- Codage (7%)
- Autre (10%)

Avantage de la COO

- Cohérence de la solution
- Facilité de communication entre développeurs (on se base sur des interfaces)
- Traçabilité du développement
- Réutilisabilité
- Maintenabilité

Inconvénient de la COO

- Tout ne se prête pas à une description objet
- Point de vue unique et éventuellement réducteur

UML 2.0

La notation UML utilise des constructions générales et abstraites valables dans différents diagrammes. UML ne prescrit pas de processus de conception.

Stéréotype :
Cette construction permet de classer des éléments du diagramme eux-mêmes lorsqu'ils sont

similaires entre eux. Elle se note entre << >> et facultativement au moyen d'une icône ad-hoc.

Valeur étiquetée

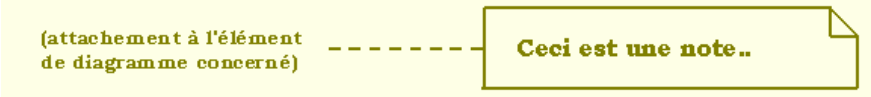
Une Tagged Value est une paire (nom,valeur) qui permet d'attacher une information arbitraire à un élément de diagramme. Elle se note entre { }.

Contrainte

Une contrainte est une relation sémantique entre 2 ou plusieurs éléments du modèle. Elle se note entre { }.

Commentaire

Un texte à l'usage du lecteur humain mais non exécutable par la machine. Il se note dans une icône particulière :



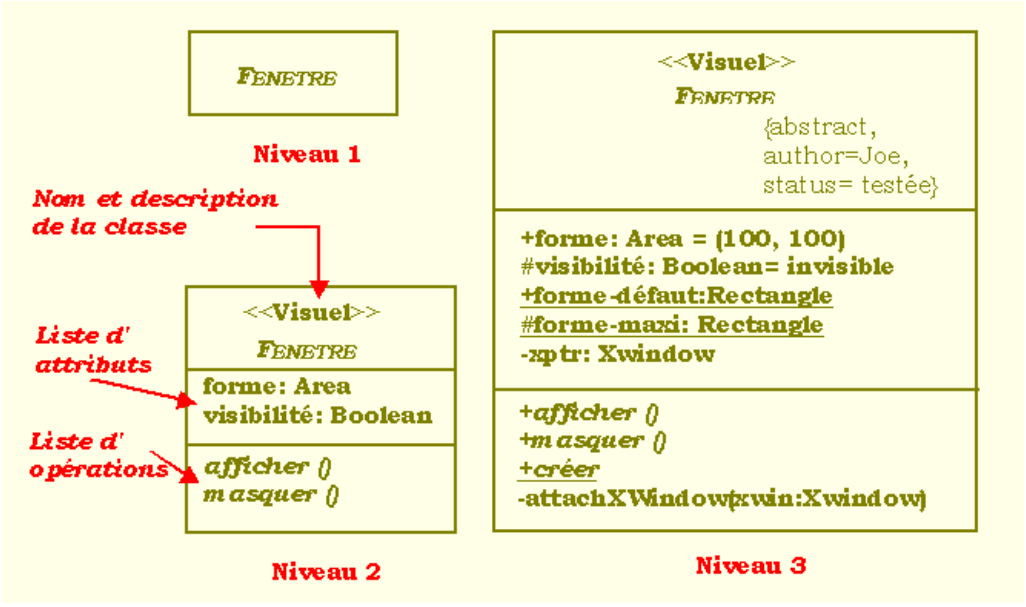
Classe

Une classe représente un concept à l'intérieur du système en cours de modélisation.

Une classe UML st un descripteur pour un ensemble d'objets similaires du point de vue : de leur structure (attributs), de leur comportement (méthodes), de leurs relations avec l'extérieur (associations, …).

La notation consiste en une boîte rectangulaire plus ou moins détaillée : au 1er niveau la boîte se réduit au nom de la classe ; Au 2ème niveau la boîte est divisée en 3 compartiments : le nom et les caractéristiques générales de la classe, les attributs, les méthodes ; Au 3ème niveau, on peut ajouter

des détails d'implémentation qui précisent la description de 2ème niveau.

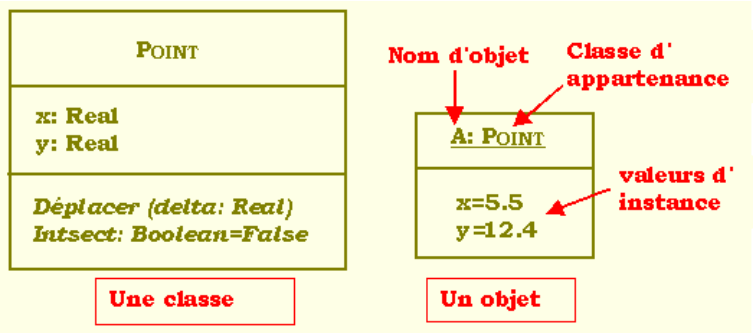


Objet

Un objet représente une instance particulière d'une classe. Il possède une identité et des valeurs d'attributs.

La question de l'identité distingue fondamentalement un objet d'un n-uplet d'une base de données :
- 2 n-uplets d'une même *table* se distinguent par la valeur d'un attribut spécial nommé clé primaire.
- 2 objets d'une même *classe* se distinguent par un identificateur externe (OID pour Object IDentifier). Il indique en fait une position mémoire attribuée par la machine, à l'exécution.

On peut formuler cette différence d'une manière imagée : en OO, 2 jumeaux sont différents par leur OID mais peuvent avoir exactement les mêmes valeurs d'attributs. En BD (relationnelle) les jumeaux sont interdits s'ils n'ont pas une propriété distinctive.



Attributs

Un attribut est un nom de donnée commun aux objets d'une classe. UML autorise 2 modes de détention :
- un nom de donnée commun mais valué spécifiquement dans chaque instance, c'est le cas général.
- un nom de donnée commun mais valué une seule fois pour toutes les instances : une valeur partagée et qui vaut pour tous les objets de la classe.

Notation : visibilité nom [multiplicité] : type= valeur_initiale {propriétés}

visibilité + visibilité publique # visibilité protégée - visibilité privée.

nom nom de l'attribut (chaîne de caractères)

[multiplicité] nombre de valeurs de l'attribut dans une instance. Par défaut, multiplicité =1 *Exemples : couleurs [3] points [2...*] texte[0...20]: String (une chaîne de caractères qui peut être vide)*

: type spécification, dépendante du type de l'attribut dans le langage d'implémentation (facultatif dans le diagramme mais pas dans le modèle d'implémentation)

valeur_initiale (facultatif dans le diagramme et dans le modèle)

{propriétés} valeurs étiquetées pour "customisation" (facultatif, donc). Par exemple {frozen} : mise à jour de valeur interdite, {obligatoire} : valuation obligatoire de l'attribut, spécification particulière d'une visibilité autre que publique, etc.

On peut également utiliser des stéréotypes. Dans ce cas le stéréotype figure en tête de la déclaration de l'attribut ou par indentation collective.

Méthodes et opérations

Une opération est la définition d'un service qui peut être demandé à une instance (objet) de la classe.

Une méthode est une implémentation d'une opération dont elle spécifie l'algorithme ou la procédure associée.

Les entrées de la boîte opération d'une classe montrent les opérations définies sur la classe et les méthodes fournies par cette classe. Opérations et méthodes possèdent une signature (nom + paramètres).

Notation : visibilité nom (liste_de_paramètres) : type_de_l'expression_retournée {propriétés}

visibilité : + visibilité publique # visibilité protégée - visibilité privée

nom nom de la méthode
(liste_de_paramètres) paramètres formels séparés par une virgule respectant la syntaxe suivante :

genre nom : type_de_l'expression = valeur_par_défaut

genre prend sa valeur dans **in, out, inout** (à défaut, genre = in)
nom nom du paramètre formel

type_de_l'expression type de l'implémentation (langage dépendant)

valeur_par_défaut facultatif et langage dépendant

: type_de_l'expression retournée type(s) d'implémentation de valeur(s) retournée(s) par l'opération (langage dépendant). Par défaut, l'opération ne retourne aucune valeur

{propriétés} valeurs applicables à la caractéristique de comportement.

Exemples :

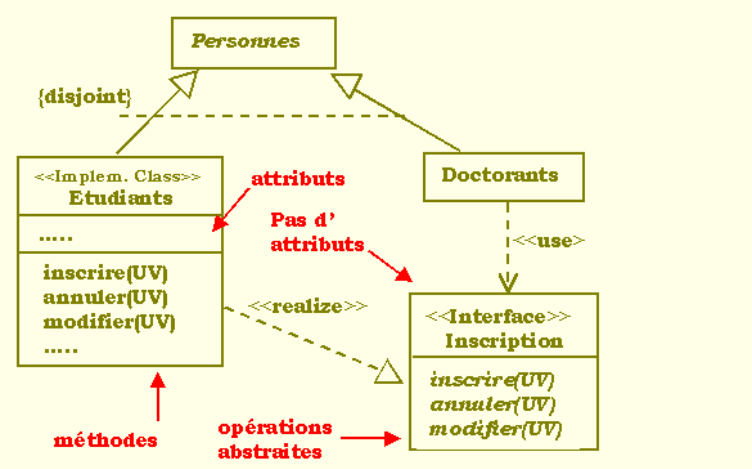
{query} : une opération sans effets de bord (ne modifie pas l'état du système d'objets) {abstract} : méthode non fournie par cette classe (notation alternative : mettre le nom de l'opération en italique).

{concurrency=nom} dans lequel nom prend ses valeurs parmi *sequential*, *guarded*, *concurrent*

La propriété selon laquelle une opération (ou méthode) concerne la classe (et non un objet) se note par le soulignement.

Les interfaces

Une interface est une classe sans attributs et dont toutes les méthodes sont abstraites. Une interface ne possède pas de constructeur



On utilise le stéréotype <<realize>> pour indiquer qu'une classe implémente une interface.

Package

Un paquetage est un regroupement d'éléments de diagramme qui entretiennent entre eux des relations étroites (on dit fortement couplés). Mais rien n'oblige un diagramme à correspondre à un seul package, ni tous les éléments d'un diagramme à appartenir à un package unique.

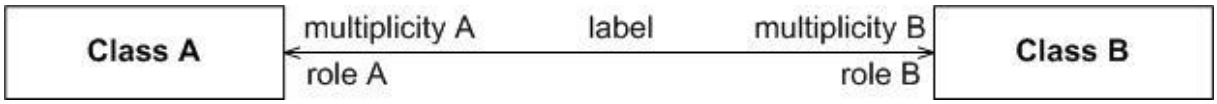
Le relations entre classes

Le clientélisme

Une classe A est cliente d'une classe B si elle utilise (déclaration d'attributs, manipulation, passage de paramètre, …) les ressources de B (attributs ou méthodes).

L'association

A est en relation d'association avec B si les instances de B sont indépendantes de celles de A. Une même instance de B peut être partagée par plusieurs instances de A. Exemple : un client et un magasin.



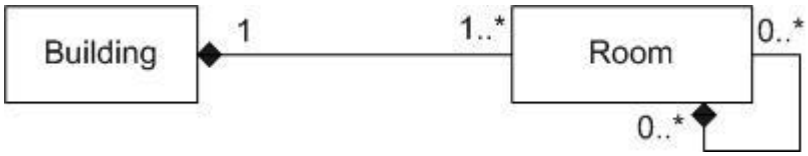
La multiplicité : 0..1 : zero ou un, 1 : un seulement, 0..* : zéro ou plus, 1..* : un ou plus, n : n seulement (avec $n > 1$), 0..n : zéro à n (avec $n > 1$), 1..n : un à n (avec $n > 1$). Elle se not à l'extrémité et non à la source.

La flèche indique la direction de l'association. Elle peut être bidirectionnelle.

Le rôle précise la place de l'objet dans l'association.

La composition

A est en relation de composition avec B si les instances de B sont dépendantes de celles de A. Une instance de B ne peut donc pas être partagée par plusieurs instances de A. Exemple : une voiture et son moteur.



L'héritage

Lorsqu'une classe B hérite d'une classe A, elle récupère toutes les méthodes et tous les attributs ayant la visibilité « public » ou « protected » (les attributs et méthodes « private » ne sont pas héritées). On peut dire qu'un objet de type B est de type A. B est une spécialisation de A. A est une généralisation (abstraction de B). En UML il n'y a pas d'héritage multiple possible.

B possède donc les mêmes méthodes que A et tout les appels réalisés sur des objets de type A peuvent être réalisés sur des objets de type B.

Exemple : Voiture hérite de Véhicule, une Voiture est un Véhicule, une Voiture spécialise un Véhicule, une Véhicule généralise une Voiture.

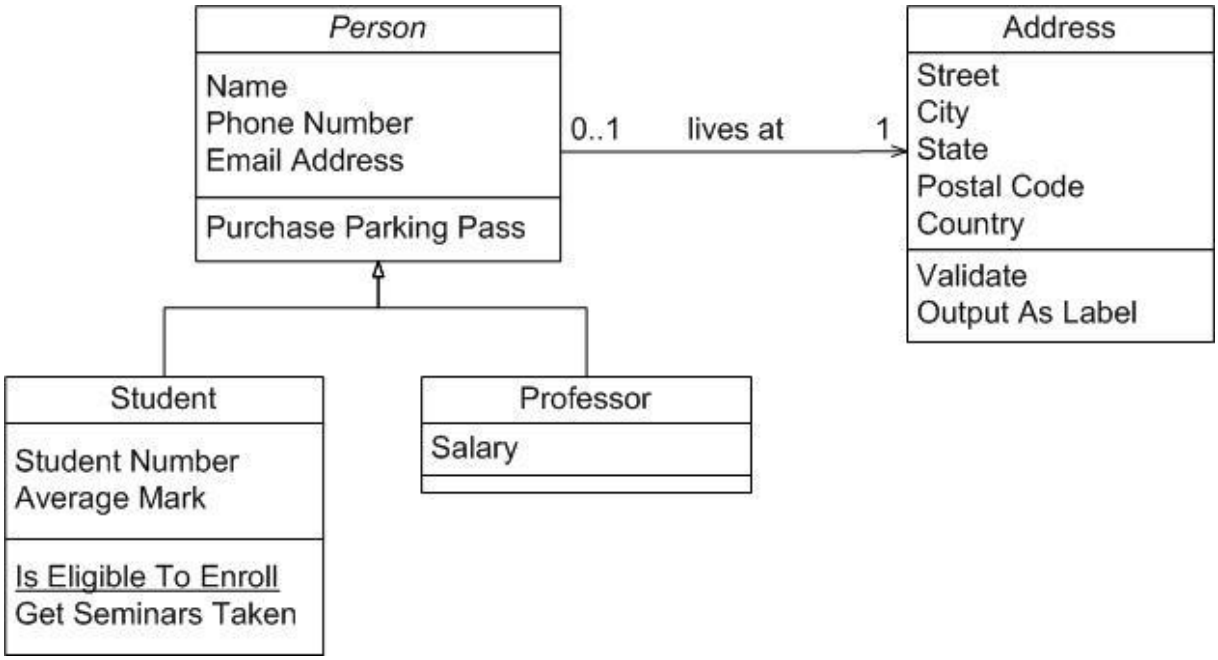
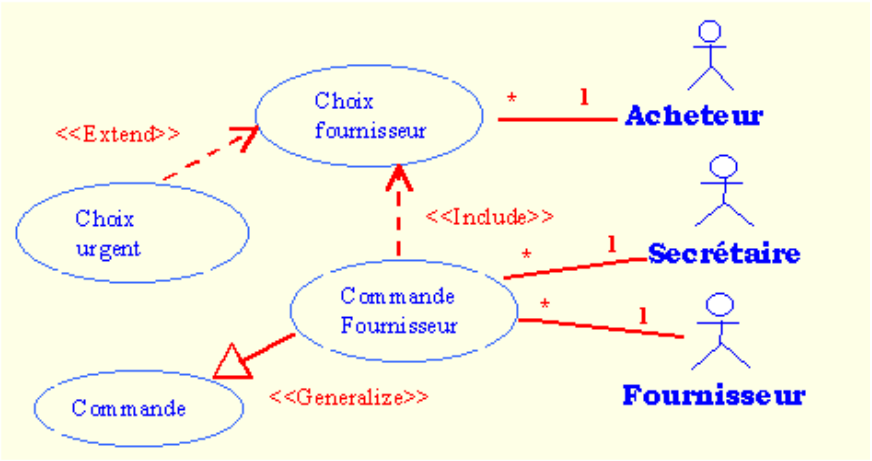


Diagramme de cas d'utilisation

Permet de lister les acteurs et les scénarios qui peuvent se dérouler dans le système



<<Association>> : c'est la seule relation autorisée entre une instance d'acteur et une instance de use case.

<<Extend>> : c'est une relation entre 2 instances de Use Case telle que A extend B signifie que le comportement d'un B peut être complété par le comportement d'un A.

<<Include>> : c'est une relation entre 2 instances de Use Case telle que la réalisation de la fonction de l'un nécessite la réalisation de la fonction de l'autre.

<<Generalize>> : exprime la relation d'héritage qui sera expliquée plus en détail à l'occasion du diagramme de structure statique.

Diagrammes statiques

Objectifs : montrer la structure statique de choses existantes (comme des classes, des types, etc...) soit : leur structure interne et leurs relations.

Ce qu'il ne montre pas : le caractère temporel de l'information, soit son aspect dynamique. Néanmoins un diagramme de classes peut contenir des descriptions réifiées d'un comportement temporel.
Par exemple, on peut montrer une classe Commandes par sa structure (date de commande, date de livraison, liste de produits, opération d'annulation, etc.).

Mais le diagramme ne montre pas la transformation dans le temps de la commande en livraison ou en facture. Néanmoins l'attribut date de livraison constitue une réification d'un comportement temporel.

Diagramme de classe

Indique les relations entre les classes. Pour le réaliser on doit s'intéresser à la définition des classes et de leurs relations.

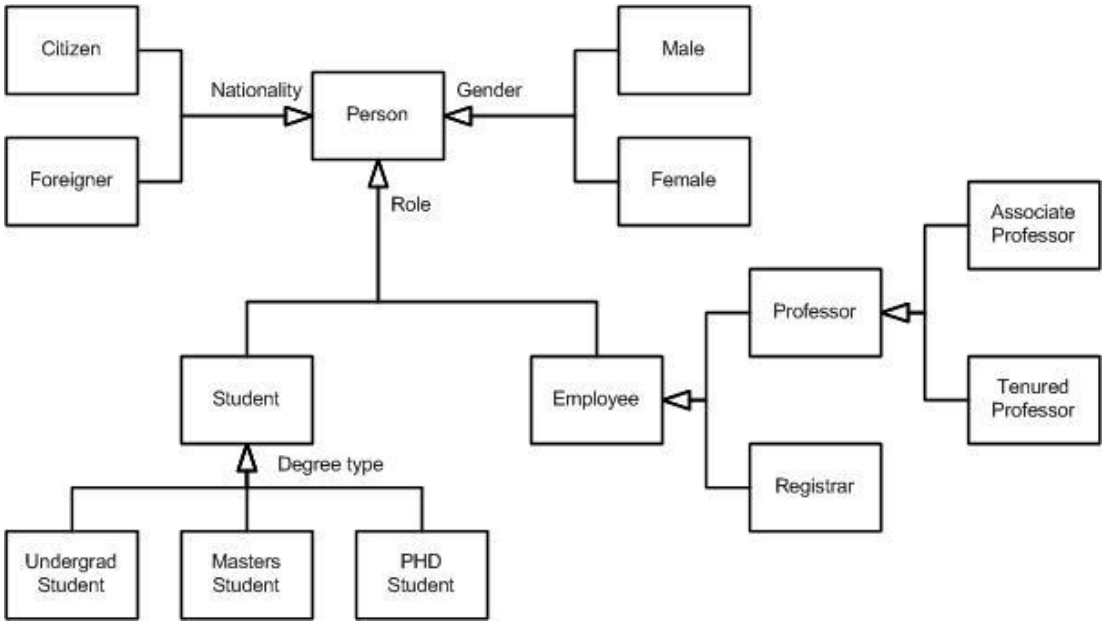


Diagramme d'objets

Un diagramme d'objets montre les instances d'une classe avec des valeurs d'attributs. Il permet d'éclairer un diagramme de classe en l'illustrant par des exemples.

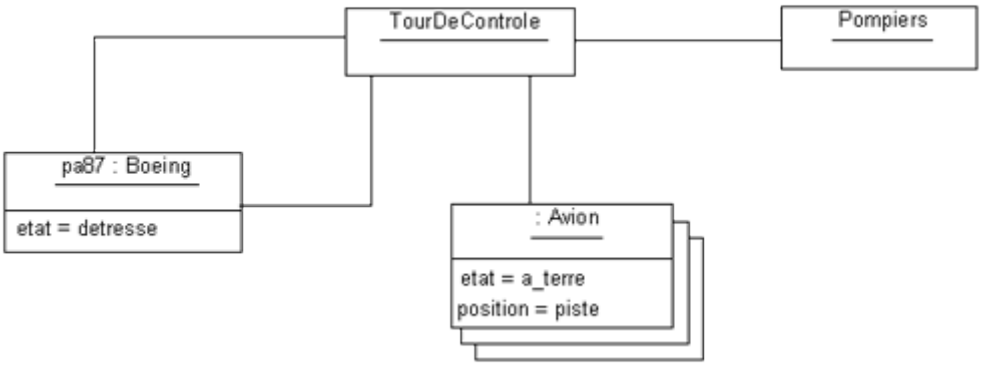


Diagramme de déploiement

C'est une vue statique qui sert à représenter l'utilisation de l'infrastructure physique par le système et la manière dont les composants du système sont répartis ainsi que leurs relations entre eux.

Diagrammes dynamiques

Diagramme de séquence

Le diagramme de séquence représente la succession chronologique des opérations réalisées par un acteur. Il indique les objets que l'acteur va manipuler, et les opérations qui font passer d'un objet à l'autre. On peut représenter les mêmes par un ***diagramme de collaboration***, graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets : diagramme de séquence et diagramme de collaboration sont deux vues différentes, mais logiquement équivalentes (on peut construire l'une à partir de l'autre), d'une même chronologie.

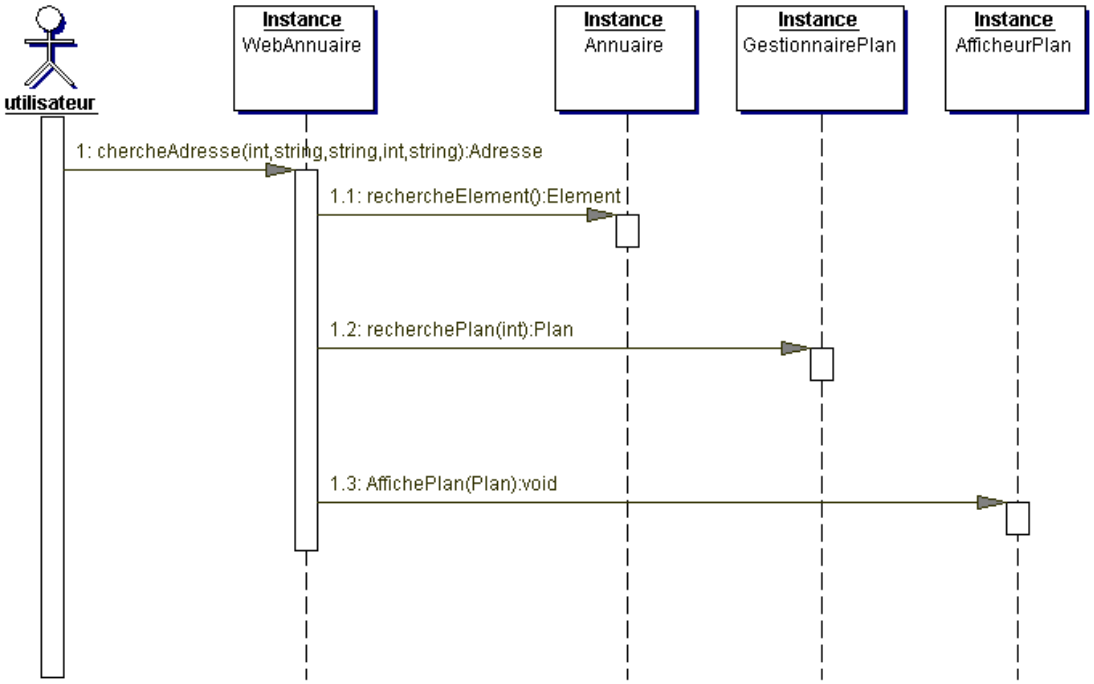


Diagramme de communication

C'est une représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets.

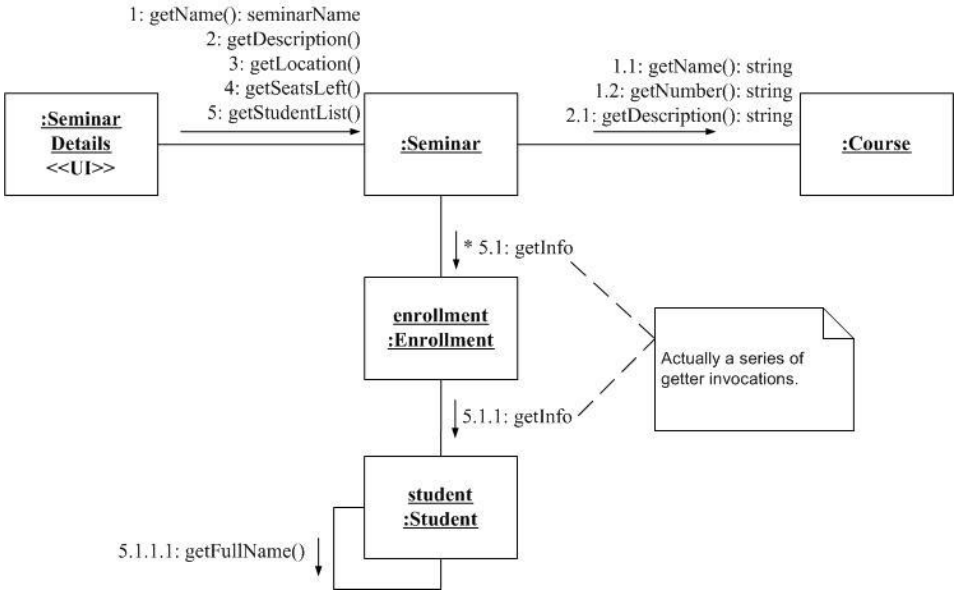


Diagramme de collaboration

Montre le flux et l'ordre des messages entre les différents objets.

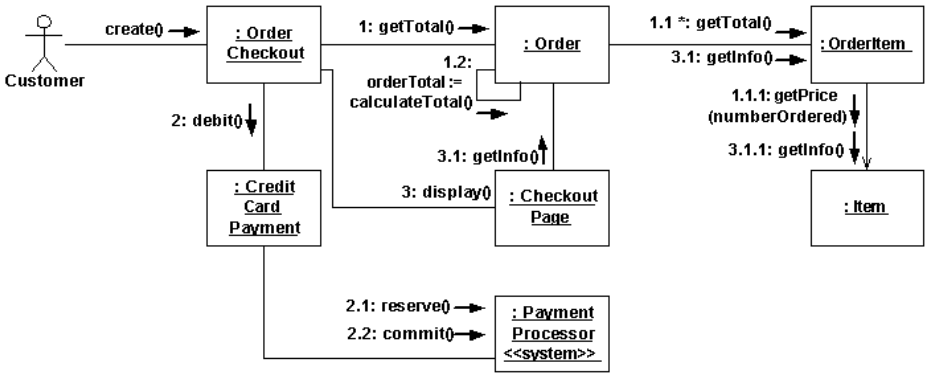


Diagramme d'activité

Il permet de décrire sous forme de flux ou d'enchaînement d'activités le comportement du système ou de ses composants.

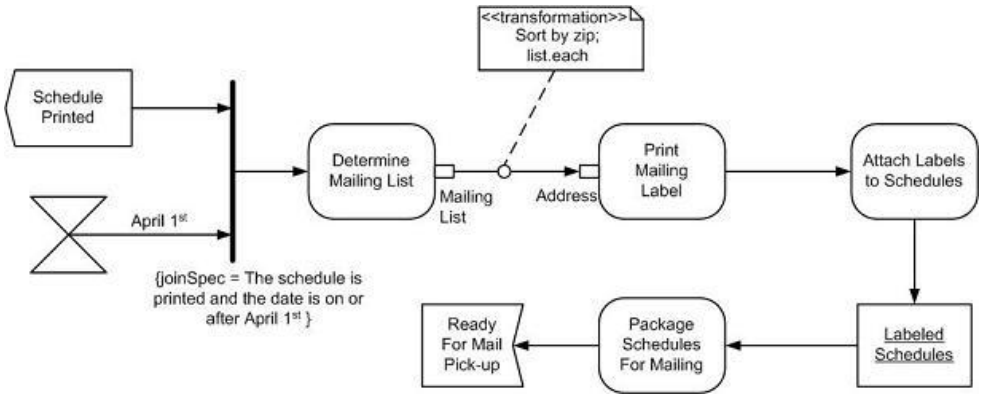


Diagramme de composant

Utilisé pour les systèmes vastes, il permet de donner l'architecture du système.

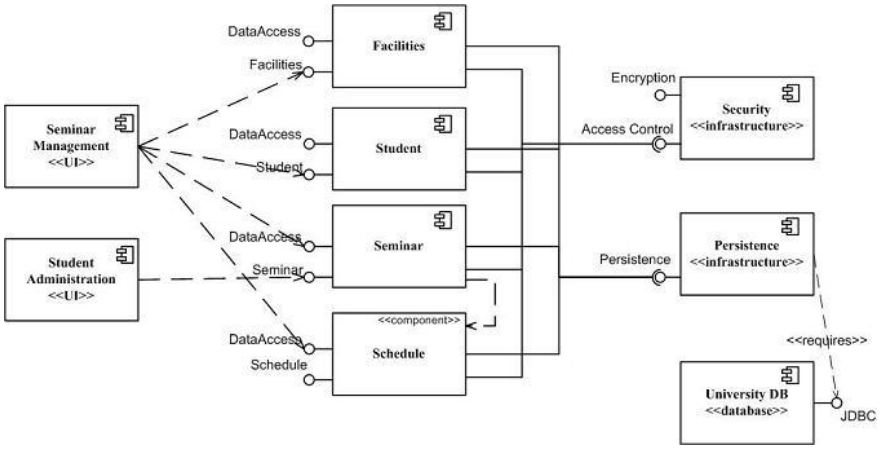


Diagramme d'états

Le diagramme d'état représente la façon dont évoluent ("cycle de vie") durant le processus les objets appartenant à une même classe. La modélisation du cycle de vie est essentielle pour représenter et mettre en forme la dynamique du système.

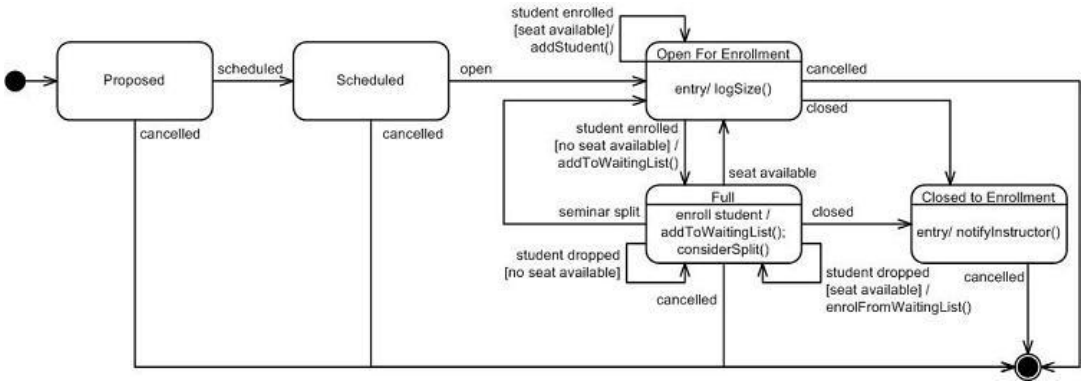


Diagramme de packages

Représente le système sous forme de paquetages.

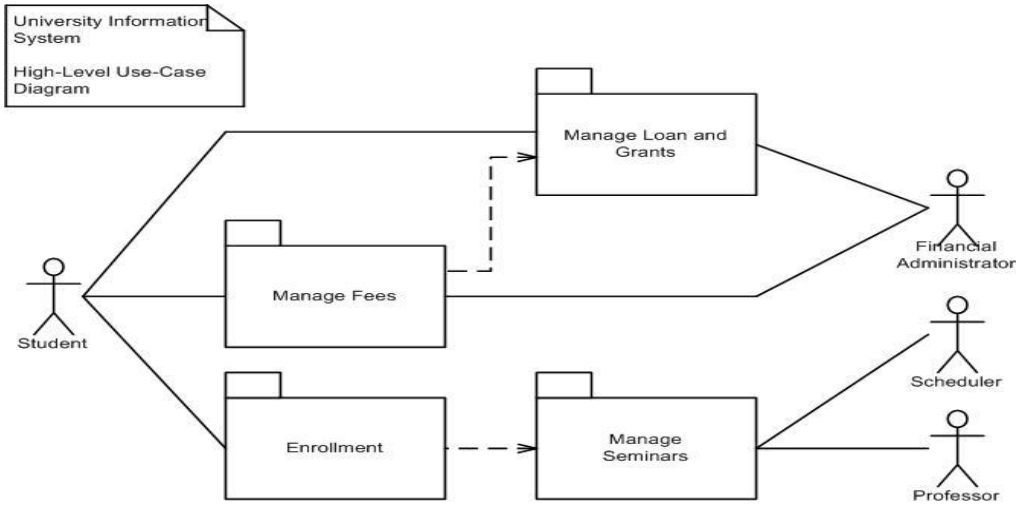


Diagramme de structure composite

…

L a programmation par contrat

La **programmation par contrat** est un **paradigme** de programmation dans lequel le déroulement des traitements est régi par des règles. Ces règles, appelées des **assertions**, forment un contrat qui précise les responsabilités entre le client et le fournisseur (une classe) d'un morceau de code logiciel. C'est une méthode de programmation semi-formelle dont le but principal est de réduire le nombre de bogues dans les programmes.
On précise ainsi pour des méthodes ou des attributs des pré-conditions, des post-conditions et des invariants.
Une méthode ne peut s'exécuter que si les pré-conditions sont remplies et en retour elle assure que les post-conditions le sont.

> Exemple : pour une fonction calculant la racine carré d'un nombre x : précondition : $x \geq 0$, postcondition : *résultat* ≥ 0 et *résultat*$^2 = x$

La pré-condition assure que le développeur utilise la fonction correctement, alors que la post-condition permet au développeur de faire confiance à la fonction.

Une fois les assertions fixées, le développeur n'a pas à vérifier que les entrées et les sorties sont correctes.

Java

Langage récent (SUN 1995), portable (notion de machine virtuelle), extensible (API)

Pas de pointeur (notion de handle).

Les outils

Javac

Permet de compiler une classe ou un programme

- **javac MaClasse.java** : compilation de MaClasse.java, génére MaClasse.class
- **javac *.java** : compilation de toutes les classes du répertoire courant

Java

Permet d'exécuter un programme java

- **java MaClasse**

Javadoc

Javadoc est un outil puissant permettant de générer la documentation des classes au format html.

Toute la documentation java (http://java/sun.com/docs) est générée grâce à ce mécanisme. Vous devez prendre l'habitude d'avoir ce lien ouvert lorsque vous développez afin de vous documenter sur les classes et les méthodes que vous manipulez.

La documentation est générée grâce à des commentaires spéciaux renseignés tout au long du code pour les attributs et les méthodes.

Voici quelques uns des mots clés utilisés :

- Les commentaires sont entourés par **/** … */**
- Chaque ligne de commentaire commence par *
- Chaque mot clé commence par **@**
- **@version** : indique la version de la classe ou de la méthode
- **@author** : indique le ou les auteurs
- **@see** : renvoie vers d'autres références
- **@inheritDoc** : indique que les commentaires de la méthode (resp. classe) parente sont inclus dans la méthode (resp. classe) courante.
- **@param Nom commentaire** : indique que l'on donne un commentaire sur un paramètre de méthode
- **@return** : commentaire sur ce que retourne la méthode
- **@since :** indique la date de début de développement du code
- **@throws** : indique que la classe ou la méthode peut renvoyer une exception

Génération de la documentation

- **javadoc MaClasse.java** : génère la documentation de MaClasse
- **javadoc –private MaClasse.java** : génère également la documentation des attributs ou méthodes privées de MaClasse
- **javadoc *.java** : génère la documentation de toutes les classes java du répertoire courant

Pour commenter un paquetage : créer un fichier `package-info.java` dans le même répertoire que le paquetage. Puis mettre les commentaires et inclure le fichier dans le paquetage

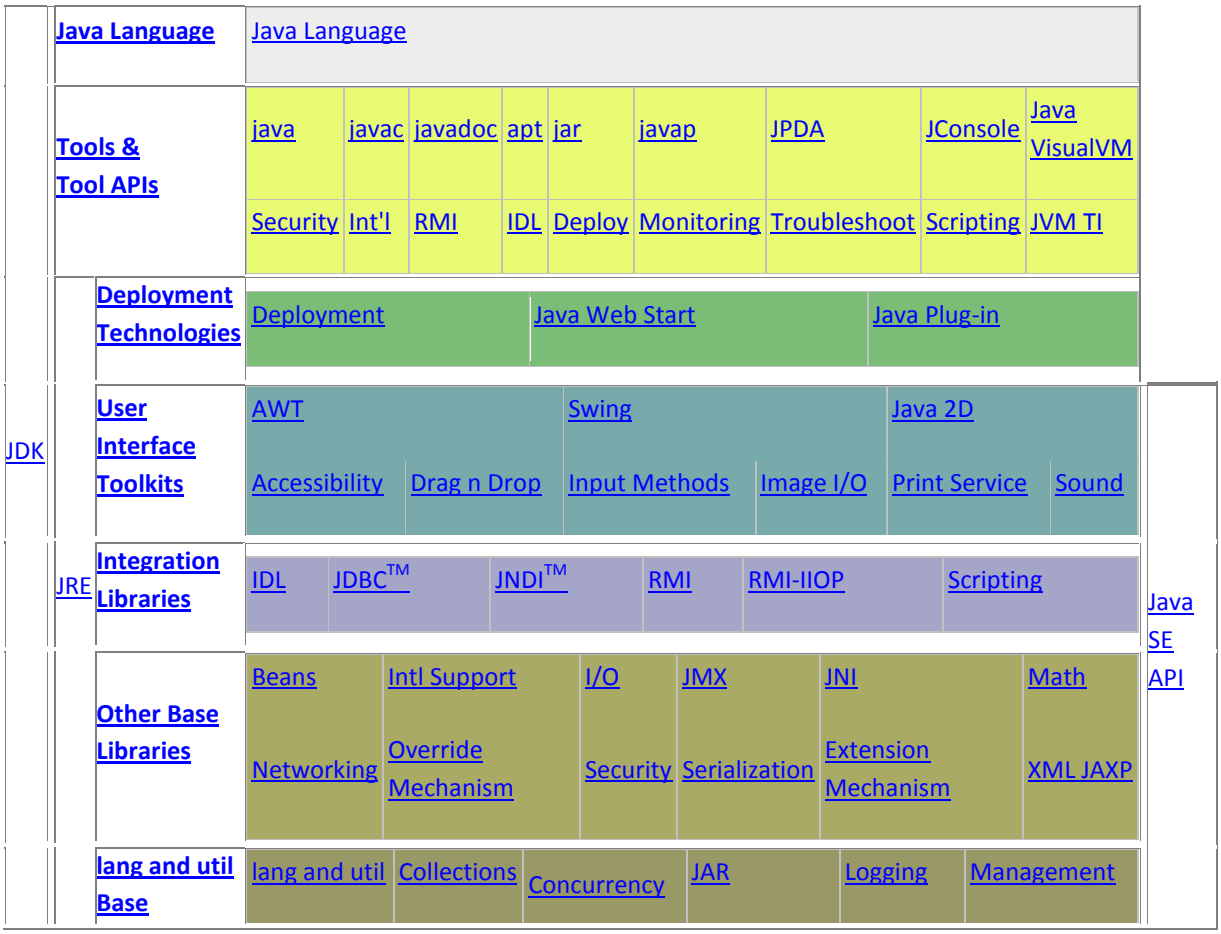
```
/**
 * Commentaires*
 * @author …
 */
package NomEtCheminDuPaquetage;
```

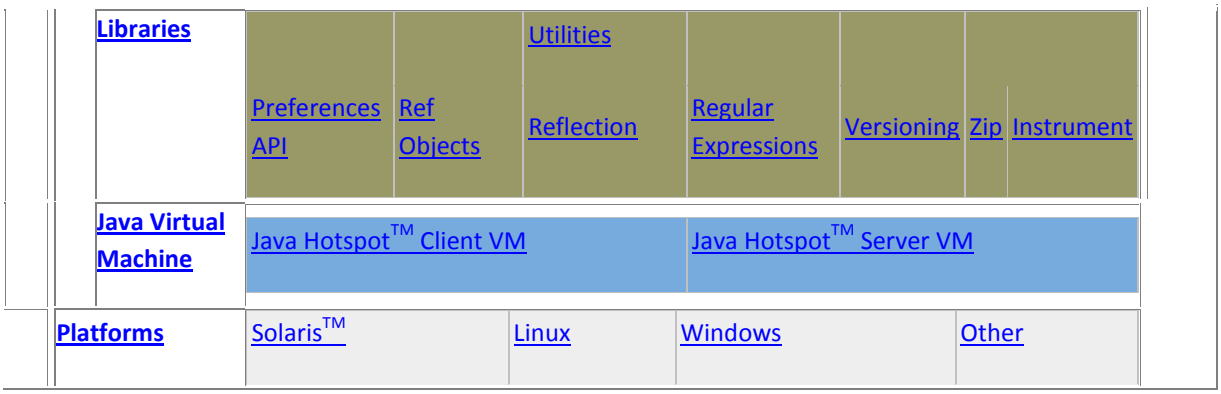
Jar

Java Archive Files. Format permettant de regrouper plusieurs fichiers.

Pour créer une archive jar : jar –cF nomfichier.jar .

Pour spécifier qu'un programme va utiliser une archive : java –cp nomfichier.jar MonAppli





Eclipse

Environnement de développement pour Java. http://www.eclipse.org/

EclipseUml

Extension UML pour Eclipse http://www.eclipsedownload.com/

Fujaba

Un outil de modélisation UML avec exportation de code java. http://wwwcs.uni-paderborn.de/cs/fujaba/

Umbrello

Un outil de modélisation UML http://uml.sourceforge.net/index.php

Les règles en java

Le typage

On trouve en Java différents types

1. Les types primitifs : int, float, char, double, … Ce ne sont pas des objets
2. Les types objet : classes prédéfinies en java ou alors classes propriétaires

Dans la suite du document « TYPE » signifiera indifféremment l'un ou l'autre, « TYPE_PRIM » signifiera le premier type et « TYPE_OBJET » signifiera le second type.

Nommage des méthodes

La convention (sans obligation) de nommage des méthodes en java est la suivante :

1. Le nom commence par une minuscule
2. S'il comporte plusieurs mots, ils sont accolés avec une majuscule pour le début de chaque mot (sauf le premier).

Les règles de visibilité

S'appliquent aux attributs et aux méthodes et sous certaines conditions aux classes

1. Publique : toutes les méthodes de toutes les classes peuvent accéder à un attribut ou une méthode déclarée avec la visibilité « **public** ». On déconseille d'utiliser la visibilité publique pour les attributs afin de garantir la cohérence des valeurs. Ceci est obligatoire si des attributs dépendent d'autres attributs. La mise à jour des attributs se fera par l'intermédiaire de méthodes spécifiques.
2. Protégée : seules les méthodes de la classe courante et de ses dérivées peuvent accéder à un attribut ou une méthode déclarée avec la visibilité « **protected** »
3. Privée : seules les méthodes de la classe courante peuvent accéder à un attribut ou une méthode déclarée avec la visibilité « **private** »

Dans la suite des documents « VISIBILITE » indique « private », « protected » ou « public ».

La méthode toString()

Il est conseillé en Java de mettre une méthode toString() dans chaque classe afin de convertir le contenu de celle-ci en chaîne de caractère pour un affichage aisé lors des phases de débogage.

Le prototype de cette méthode est : **public class toString() {….}**

Les classes

L'entête de la classe

{public | private} class MaClasse { ….}

Les attributs

La déclaration des attributs est faite au même endroit (début ou fin de la classe) afin de faciliter la lecture. Ils sont de préférence regroupés en fonction de leur règle de visibilité.

Déclaration : **VISIBILITE TYPE Nom ;**

Les méthodes

Les méthodes sont déclarées de la manière suivante : **VISIBILITE TYPE NOM (PARAMETRES) {CORPS}**

« PARAMETRES » étant la liste des paramètres de la méthode ayant chacun un « TYPE » et un Nom.

Exemple : public void Afficher() {…} ou private int calculerPerimetre(Figure F) { …}

Constructeur

Les constructeurs sont des méthodes portant le même nom que la classe et appelées (en fonction des paramètres d'entrée) lors de l'instanciation de l'objet

Accesseur

Les accesseurs sont des méthodes qui permettent d'accéder aux attributs de la classe ou de calculer des valeurs sans modifier ceux-ci.

Modifieur

Les modifieurs permettent de mettre à jour, directement ou indirectement, des attributs.

Abstraite

Méthode dont on connaît uniquement le prototype (sans code) : **VISIBILITE abstract TYPE MaMéthode(PARAMETRES) ;**

Surcharge

Plusieurs méthodes peuvent porter le même nom, on dit qu'il y a surcharge des méthodes. Ces méthodes diffèrent par leurs paramètres afin de savoir quelle méthode exécuter lors d'un appel.

Accès

Objet.methode() ;

Les classes abstraites

Une classe abstraite est une classe possédant au moins une méthode abstraite

{public | private} abstract class MaClasse { ….}

Déclaration d'une instance de classe

Déclaration comme les attributs : **VISIBILITE MaClasse MonObjet ;**

Seules les classes ayant la visibilité « public » peuvent être déclarées

Instanciation

MonObjet= new MaClasse(…) ;

Cette instruction fait appel à un constructeur de la classe MaClasse.

Seules les classes ayant la visibilité « public » et non abstraites peuvent être instanciées

Exemples

public class Voiture { public String Marque ; public int prix ; … }

Les classes anonymes

Une classe anonyme est une classe interne à une autre, elle ne peut être utilisée que dans la classe englobante.

Les interfaces

{public |private} interface NomDeLinterface {… }

Les relations entre classes

On utilise le mot clé **« extends »** pour indiquer un héritage. Une classe ne peut hériter que d'une seule autre classe (concept orienté objet).

{public | private} class MaClasse extends MaClasseMère { …}

Une interface peut également hériter d'une autre interface avec le même mot clé.

Une classe peut implémenter plusieurs interfaces. C'est le même principe que l'héritage sauf que le corps des méthodes n'est pas codé dans l'interface donc pour que la classe ne devienne pas abstraite, il faudra qu'elle implémente toutes les méthodes héritées.

En java on utilise le mot clé **« implements »**

{public | private} class MaClasse extends MaClasseMère implements Interface1, Interface2 { …}

Le polymorphisme

Lorsqu'une classe B hérite d'une classe A, on peut redéfinir certaines méthodes. Ces méthodes ont exactement le même prototype, seul le code change.

Lors d'un appel, la méthode exécutée sera la méthode la plus spécialisée. Le choix est fait dynamiquement en fonction du type réel de l'objet et non du type de déclaration.

Si B hérite de A alors on peut faire les affectations suivantes :

B b = new B() ; A a =new A() ; a=b; A a = new B();

mais pas l'affectation suivante : b=new A();

Si une méthode m() est définie dans A alors elle le sera automatiquement dans B (héritage). Si m() n'est pas redéfinie dans B alors :

B b = new B() ; b.m() ; et A a = new A() ; a.m() ; donneront la même exécution.

Si m() est redéfinie dans B alors

A a = new A() ; a.m() ; exécutera la méthode m() de A.

B b = new B() ; b.m() ; exécutera la méthode m() de B

A a = new B() ; a.m() ; exécutera la méthode m() de B (polymorphisme : c'est la méthode du type réel de a qui est utilisée).

Le casting

Lorsque l'on veut utiliser les fonctionnalités d'un objet instancié avec un type dérivé on utilise le casting.

Exemple : B hérite de A. A a = new B() ;

On peut faire : ((B)a).UneMethodeDeB() ;

Les paquetages

Déclaration

package NomEtCheminDuPaquetage ;

Utilisation

Import NomEtCheminDuPaquetage ;

La libération de la mémoire

La politique de java est de laisser la machine virtuelle gérer la mémoire et libérer celle-ci lorsque des objets ne sont plus référencés.

On appelle ce mécanisme le ramasse miette (garbage collector). Il se met en route seul lorsque le programme est inactif ou alors lorsqu'il consomme trop de mémoire. Cette étape est donc transparente pour le développeur. Lorsque le ramasse miette est appelé, la méthode finalize() de chauqe classe (si elle existe) est appellée.

On peut lancer le ramasse miette à la main à l'aide la ma méthode « System.gc() ; ». Pour connaître l'état de la mémoire, on peut utiliser les instructions suivantes.

```
Runtime r = Runtime.getRuntime();
// Mémoire non allouée
long free = r.freeMemory();
// Mémoire totale
long total = r.totalMemory();
```

// Mémoire maximale que la machine virtuelle peut allouer
long max = r.maxMemory();

Entête de la méthode finalize
protected void finalize() throws Throwable
Il est conseillé d'appeler la méthode parente dans la méthode finalize().
On ne peut pas prédire quand l'appel à finalize sera effectué dans la mesure où le ramasse miette fonctionne en tâche de fond. Si on veut que les méthodes finalize() soient exécutées à la terminaison du programme, il faut exécuter : System.runFinalizationOnExit(true) ;

Le point d'entrée du programme : main

La méthode main est le **point d'entrée** dans le programme.

Son prototype est toujours le même : **public static void main(String[] args) { …}**

Args est la liste des arguments passés au programme.

Les entrées sorties standards

Librairie « System », classes « out » et « in ».
La classe « out » possède entre autre les méthodes print() (affichage sans retour à la ligne) et println() (affichage avec retour à la ligne).
Ces méthodes fonctionnent avec tous les types primitifs et toutes les classes ayant une méthode toString().
On trouve également depuis la version 1.5 le printf façon langage C (exemple : printf("la valeur est %d", 1);)
L'addition permet de concaténer des chaînes de caractères.
Exemples : int i=1 ; String S = "la valeur de i est » ;
System.out.println(S+i); // affiche "la valeur de i est 1 »

La classe Scanner : elle permet de lire au clavier (ou sur tout autre entrée)
Scanner s = new Scanner(System.in); // declaration d'un scanner sur le clavier
x = s.nextDouble(); // lecture d'un double sur le clavier

Accès à l'objet courant

On utilise le mot clé « this ».

Exemple :
public class A{

```
private int I;
void setI(int i) {
        this.i =i ; // met la valeur du paramètre « i » dans l'attribut « i ». } }
```

Méthode parente

Dans une méthode, lorsqu'on utilise le mot clé super, on accède aux méthodes de la classe parente.

Exemple : dans le constructeur « super(param) ; » appel le le constructeur de la classe parente. Dans une autre méthode « super.afficher() ; » appelle la méthode afficher() de la classe parente.

Static

Le mot clé static est utilisé pour définir des attributs ou des méthodes dites de classe.

Un attribut de classe est un attribut commun à toutes les instances de la classe.

Déclaration : public static int i ;

Une méthode de classe ne s'applique pas sur un objet, elle ne le modifie pas et elle n'accède qu'à des méthodes et attributs « static ».

Déclaration : public static type methode(Param){…}

On peut accéder à une méthode de classe sans avoir déclaré d'instance de cette classe.

MaClasse.MaMethodeStatic() ;

Final

Pour empêcher la modification d'un attribut, la redéfinition d'une méthode ou la dérivation d'une classe, on utilise le mot clé final.

Exemples :

- public final int i=12; // on ne peut modifier la valeur de i donc i est une constante valant 12.
- public final Voiture v=new Voiture() ; // on ne peut modifier la valeur de v (le handle) mais on peut modifier le contenu des attributs référencés (ex : v.changerMarque(« peugeot ») ;)
- public final void afficher() {….} // indique que la méthode ne pourra pas être redéfinie dans une classe dérivée
- public final class MaClasse {…} // MaClasse ne pourra pas être dérivée

Les constantes

Pour déclarer une constante, déclarer un attribut en « static final »

Récupération du type réel de la classe

On utilise « instanceof »

Si « a » est de type « A » : « a instanceof A » renvoie « true »

Si « b » hérite de « A » : « b instanceof A » renvoie « true »

A utiliser avec modération, préférez le polymorphisme.

La comparaison

==

Teste l'égalité entre deux références (handle)

equals

a.equals(b) : execute la méthode equals() de l'objet. Cette méthode est automatiquement définie pour toutes les classes. Si on ne code pas cette méthode pour une classe particulière, elle fonctionne comme ==

compareTo

a.compareTo(b) : les classes qui implémentent l'interface « Comparable » doivent coder cette méthode. Cette méthode permet de comparer des objets. Les objets doivent pouvoir être ordonnés (plus petit, égal, plus grand). On peut être amené à utiliser la version générique de Comparable (cf. généricité).

public int compareTo(Object o) { …}

si l'objet courant est plus petit que l'objet en paramètre, la méthode renvoie -1, si il y a égalité elle renvoie 0, sinon elle renvoie 1

compare(a,b)

Compare la valeur de deux objets. La classe doit implémenter l'interface « Comparator ». Utilisé pour des structures complexes.

Les exceptions

La gestion des erreurs en Java est réalisée à l'aide des exceptions. Une exception est un objet, elle est donc associée à une classe et on peut définir ses propres exceptions (exemple : public class MonException extends Exception {…})

Déclaration et instanciation d'une exception : MonException e = new MonException() ;

Levée d'une exception : throw e ;

Lorsqu'on lève une exception, on sort du bloc courant (méthode, …) et l'exception est propagée au niveau supérieur jusqu'à ce qu'elle soit récupérée ou que l'on sorte du programme.

Récupération et traitement d'une exception :

```
try { // code
} catch(MonException e) {
// Traitement
}
catch(MonException2 e) {
// Traitement
}
finally {
}
```

Les blocks catch sont examinés les uns après les autres et celui correspondant à l'exception levée est exécuté. L'ordre a donc une importance. On commence par les blocks les plus spécifiques avant de mettre les blocks généraux (traitement de l'exception Exception par exemple).

Le block finally est toujours exécuté lorsque l'on sort du try.

Lorsqu'une méthode peut générer une exception (c'est-à-dire si elle n'est pas récupérée dans la méthode), il faut l'indiquer dans l'entête de la méthode

public void maMethode() throws MonException { …]

Clonage et copie

Dans la classe Object, une méthode clone est définie. Elle doit être redéfinie dans chaque classe qui peut être clonée.

La classe clonable doit implémenter l'interface « Cloneable ».

Entête de la méthode clone : public Object Clone() {…}

Le clonage en surface (copie bit à bit de l'objet)

Public Object Clone()

```
{       Object c = null;
        try { c = super.clone(); }
        catch (CloneNotSupportedException e){ }
        return c;
}
```

Le clonage en profondeur

On recopie les objets référencés.

```
public Object clone() throws CloneNotSupportedException
{       Object c = null;
        try { c = super.clone(); }
        catch (CloneNotSupportedException e){ }
        ((MaClasse)c).Attr1 = ((MaClasse)Attr1).clone() ;
        ….
        return c;
}
```

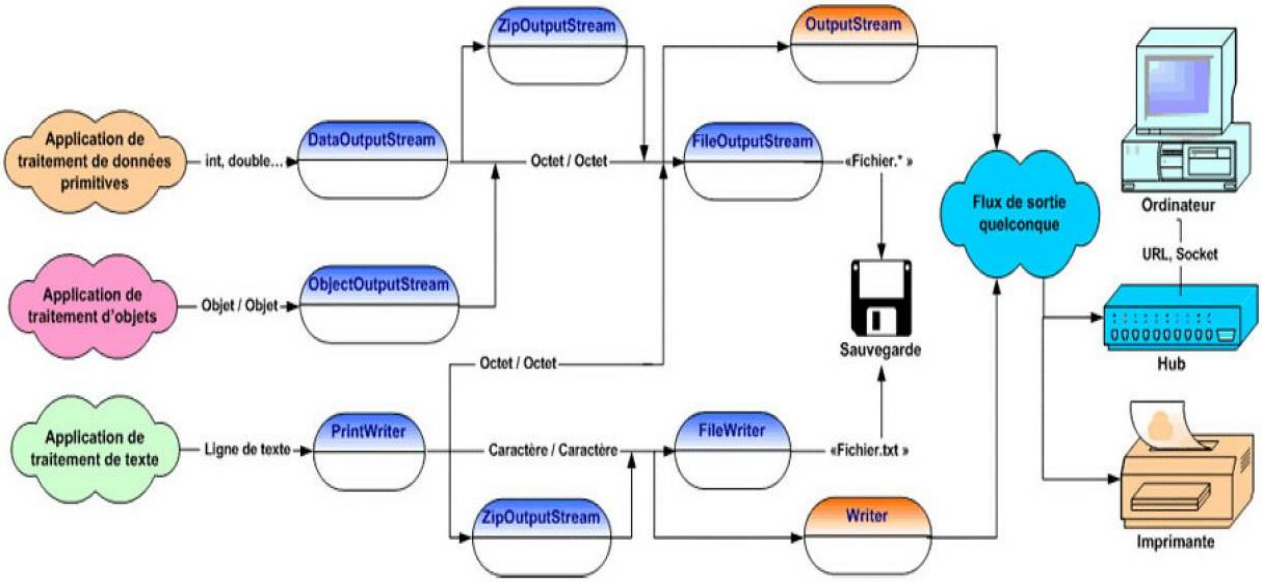
On peut également utiliser sa propre méthode de copie sans passer par ce schéma mais c'est déconseillé.

Type objet et type primitif

Il y a conversion automatique entre les types primitifs et leur représentation objet.

Exemple : int i = 27 ; Integer i2 = new Integer(14); i=i2;

Sérialisation



La sérialisation permet de convertir un objet en suite d'octet de manière à le stocker sur disque, l'envoyer sur le réseau, ….

Cela permet d'implémenter la persistance des objets (existence de l'objet après la terminaison d'un programme).

L'interface Serializable

Les objets pouvant être sérialisés doivent implémenter l'interface « Serializable »

Dans le cas de la sérialisation d'objets, il faut déclarer un OutputStream qui est le flux sur lequel l'objet sera envoyé (fichier, url réseau, …) et un ObjectOutputStream qui est la conversion de l'objet en octets. On utilise ensuite la méthode writeObject() de l'ObjectOutputStream pour écrire les objets sérialisable vers le flux.

Pour réceptionner un objet, on déclare un InputStream qui est le flux entrant à partir duquel sera réceptionné l'objet puis un ObjectInputStream qui va permettre de convertir le flux d'octets en un objet. On utilise la méthode readObject() de l'ObjectInputStream afin de lire l'objet.

Exemple :

```
Public class A implements Serializable {…}
A a = new A();
OutputStream f = new FileOutputStream("unfichier");
ObjectOutputStream out = new ObjectOutputStream(f);
out.writeObject(a);
out.close();

Public class A implements Serializable {…}
A a ;
```

```
InputStream f = new FileInputStream("unfichier");
ObjectInputStream in = new ObjectInputStream(f);
a = (A)in.readObject();
in.close();
```

La sérialisation est automatique et tous les objets sérialisables sont convertis.
Dans le cas des types primitifs, on peut soit utiliser la sérialisation des objets, soit utiliser un DataOutputStream.
Dans le cas de la sérialisation des chaînes de caractères, on peut utiliser PrintWriter et FileWriter.

Remarques :

1. si la classe dérivée de Serializable implémente les méthodes :

- private void writeObject(ObjectInputStream s) throws IOException
- private void readObject(ObjectOutputStream s) throws IOException

celles-ci sont appelées pour sauvegarder ou lire les champs propres de la classe (pas ceux de la classe héritée) plutôt que les méthodes par défaut. Ceci peut servir pour sauvegarder des informations complémentaires ou sauvegarder des champs non sérialisables.

Les méthodes suivantes sont équivalente à la sauvegarde par défaut :

```
private void writeObject(ObjectInputStream s) throws IOException{
  s.defaultWriteObject();
}
private void readObject(ObjectOutputStream s) throws IOException {
    s.defaultReadObject();
}
```

2. Chaque fois qu'un objet est sauvé dans un flux, un objet *handle*, unique pour ce flux, est également sauvé. Ce *handle* est attaché à l'objet dans une table de *hashage*. Chaque fois que l'on demande de sauver à nouveau l'objet, seul le *handle* est sauvé dans le flux. Si on veut sauvegarder une nouvelle version de l'objet, il faut utiliser la méthode **writeUnshared().**
3. Il est conseillé d'ajouter un attribut **static final long serialVersionUID** dans les classes sérialisables indiquant la version de la classe afin de garantir la cohérence des objets.
4. Si on veut éviter qu'un attribut soit sérialisé, il faut utiliser le mot clé **transient** placé devant l'attribut**.**

L'interface Externalizable

Si on veut contrôler la persistance, on implémente l'interface « Externalizable ».

Lorsque l'on implémente cette interface, deux méthodes doivent être déclarées.

```
public void readExternal(ObjectInput in) throws
IOException,ClassNotFoundException { ….}

public void writeExternal(ObjectOutput out) throws IOException { …}
```

On peut donc coder la manière dont la persistance est gérée. La différence avec la méthode précédente réside dans le passage des paramètres.

Exemple

```
Public class A implements Externalizable {…}
A a = new A();
OutputStream f = new FileOutputStream(“unfichier”);
ObjectOutputStream out = new ObjectOutputStream(f);
a.writeExternal(out);
out.close();
```

```
Public class A implements Externalizable {…}
A a ;
InputStream f = new FileInputStream(“unfichier”);
ObjectInputStream in = new ObjectInputStream(f);
a .readExternal(in);
 in.close();
```

Remarques

Il est possible de compresser les données sérialisées en utilisant les classes ZipOutputStream et ZipInputStream (java.util.zip.*).

La généricité

La généricité est présente en java depuis la version 1.5. Plus sûre et plus rapide (contrôle effectué à la compilation par le compilateur) que l’héritage mais beaucoup moins souple (réduit le polymorphisme).

Elle permet de créer des classes, des interfaces ou des méthodes qui sont paramétrées par un type.

Exemple : une paire de deux éléments de même type

```
Public class Paire <T> {
        Public T x,y ;
        Paire(T x, T y) { this.x = x ; this.y = y ; }
        …
        }
```

Utilisation de la paire : Paire<String> p = new Paire<String>(« un », « deux ») ;
Paire<Voiture> v = new Paire<Voiture>(new Voiture(), new Voiture()) ;

Déclaration

Class maClasse<T1, T2> { …} On peut hériter d’une classe générique.
Public <T> void maMethode(maClasse<U,Integer> c) { …}

Généricité et héritage

maClasse<String> M = new maClasseDer<String>(); // Ok si maClasseDer hérite de maClasse
par contre :
maClasse<MonType> M = new maClasseDer<MonTypDer>(); // interdit meme si MonTypeDer hérite de MonType
Le type doit être le même.

Généricité et polymorphisme

On peut forcer l'utilisation d'une méthode <String>maMethode() ;

Les types contraints

On peut imposer qu'un type générique hérite d'une classe ou implémente une interface : mot clé **extends**.
Class maClasse<T extends uneInterfaceOuuneClasse> {…}
Il peut y avoir plusieurs contraintes qui sont séparées par des **&**, si il y a une classe parmi ces contraintes, elle doit être unique et placée en tête.

Le jocker : ?

Désigne un type paramètre fixé mais inconnu au moment de la déclaration.

maClasse<? extends MonType> M = new maClasseDer<MonTypDer>();

Les structures de données

Les tableaux

Déclaration : type[] unTableau ; ou type unTableau[] ;

Multi-dimensions : int [][]a = {{1,2,3}, {4,5,6} } ;

Figure [][][] F = new Figure[2][3][4] ;

Figure [][]F = new Figure[10][] ; F[0] = new Figure[5];

F[i][j] = new Point();

Ceci déclare un tableau de références vers des objets

Instanciation : unTableau = new type[N] ;

ou pour les types primitifs on peut instancier de la manière suivante : int []a={1,2,3}

Accès aux éléments : unTableau[1] = new type() ; unTableau[1] ;

Java protège contre les débordements lors de l'accès aux tableaux. L'accès se fait de l'élément 0 à length-1

Chaque tableau a un attribut « length » qui indique la taille du tableau.

Avantage : accès rapide aux éléments, typage des éléments (on peut mettre le type même ou une de ses dérivées).

Inconvénient : taille fixe

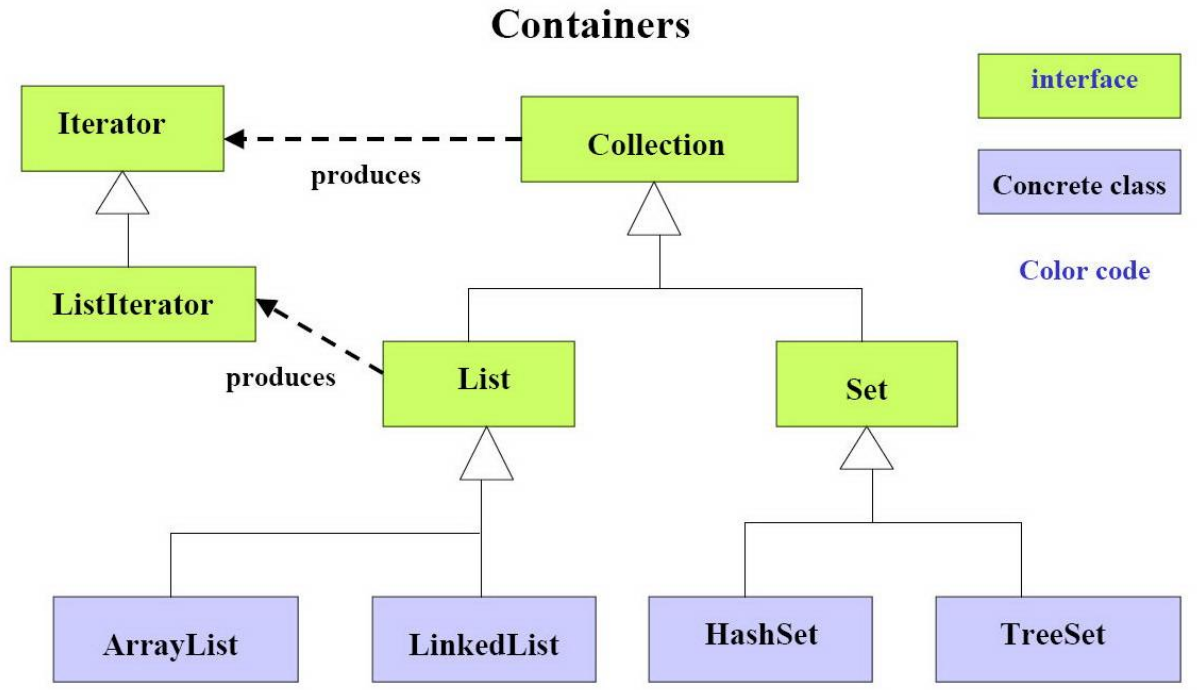
Les classes Array et Arrays

Elles proposent des méthodes permettant la manipulation des tableaux.

equals(), fill(), sort(), binarysearch(), asList() : conversion en liste,

Exemple : Arrays.fill(a,3) : rempli le tableau « a » avec des 3.

Arrays.fill(a,4,8,3) : rempli le tableau « a » des cases 4 à 8 avec des 3.



Interface collections

Une collection (ou conteneur) est un ensemble d'objets qui peuvent être soumis à des contraintes (ordre (liste), entrées uniques (ensemble), …). On sépare ici clairement la notion de collection de l'implémentation (file, pile, liste, arbre, …) ce qui met en œuvre le polymorphisme.

Pour les collections ordonnées, il faut que les objets stockés implémentent les interfaces « Comparable » ou « Comparator ».

Le choix de la collection est important car il conditionne les performances et les possibilités mises à disposition.

Méthodes :

Int size(), boolean isEmpty(), Boolean contains(Object element), Boolean add(Object element), Boolean remove(Object element), Iterator iterator(), Boolean containsAll(Collection c), boolean addAll(Collection c), Boolean removeAll(Collection c), Boolean retainsAll(Collection c), void clear(), Object[] toArray(),

On peut passer une collection existante au constructeur.

La classe Collections

C'est un ensemble de méthodes statiques permettant de manipuler les collections.

Static int binarySearch(List ist, Object key), Static int binarySearch(List ist, Object key, Comparator C), static void copy(List d, List S), static void fill(List l,Object o), static Object max(), reverse, min(), …

Le parcours de collections

Les iterateurs : iterator

Un itérateur est une classe permettant de parcourir une collection sans connaitre sa représentation interne (Liste, file, tableau, …).
Méthodes : boolean hasNext(),Object next(), void remove()

L'interface ListIterator

Extension de Ietraor permettant de parcourir les listes dans les deux directions, obtenir la position courante, modifier la liste.
Boolean hasPrevious(), Object previous(), int previousIndex(), void set(Oject o), void add(Object o)

Depuis Java 1.5

Collection C = …
for (Object o : C) { … } : se traduit par « pour tout objet o de la collection C »

L'interface Set

Collection sans duplications.
Deux implémentations : HashSet (stockage dans une table d'association) et TreeSet (stockage dans un arbre binaire).

L'interface List

Collection ordonnée pouvant contenir des duplications.

Les principales méthodes : Object get(int index), Object set(int index,Object e), void add(int index, Object e), Object remove(int index), abstract boolean addall(int index, Collection c), int indexOf(Object o), int lastIndexOf(Object o), ListIterator listIterator(), ListIterator subListFrom(int index),

La classe ArrayList

C'est une implémentation d'un tableau dans une liste. Le tableau est extensible et les insertions dans le tableau sont aisées et transparentes. On ne peut stocker que des objets, pas de types primitifs.

Utilisation :
ArrayList A = new ArrayList(7) ;
ArrayList<Type> nom = new ArrayList<Type>() ;
Méthodes : size(), add(Object o), add(index i, Object o) , clear(), contains(Object o), get(index i), toArray(), …

Classe Vector

Classe de collection qui engendre un tableau d'objets. Elle gère une taille dynamique de tableau.
Méthodes : add(Object o), get(Object o), remove(int index)

Classe Hashtable

Classe de collection qui gère un ensemble de couples (clé, valeur)

Méthodes : put(Object key, Object value), get(Object key), remove(Object key)

Classe LinkedList

Implémentation d'une liste chaînée.

L'interface Map

Association entre une clé et une valeur. Pas de clé dupliquées, une seule valeur associée à une clé.
Object put(Object key, Object value), Object get(Object key), Object remove(Object key), boolean containsKey(Object key), boolean containsValue(Objetc value), int size(), boolean isEmpty(), void putAll(Map t), void clear(), …

Classe SortedSet

Version triée (ordre croissant) des Set., les éléments insérés doivent implémenter l'interface Comparable.
SortedSet subSet(Object fromElement, Object ToElement), SortedSet headset(Object ToElement), SortedSet tailSet(Object FromElement), Object first(), Object last()

Classe SortedMap

Version triée des Map.

Classe Stack

Implémentation d'une pile

Classe TreeMap

Implémentation qui s'appuie sur un arbre bien équilibré d'une table de correspondance.

Les structures de données génériques

On peut imposer que dans une collection on ne mette pas de Object mais des instances d'une certaine classe (contenu homogène).

Comparaison

Sans généricité

```
ArrayList maList = new ArrayList() ;
// Ajout
```

Avec généricité

```
ArrayList<Personne> personnes = new ArrayList<Personne>() ;
// Ajout
```

```
maList.add(new Voiture()) ;
maList.add(new Personne()) ;
// Extraction
Voiture v =(Voiture)maList.get(0) ;
Personne p =(Personne)maList.get(1) ;
```

```
personnes.add(new Personne());
personnes.add(new Voiture()) ; // Erreur
// Extraction
Personne p = personnes.get(0) ;
```

Le réseau

On utilise les sockets à travers l'API java.net

Côté serveur

```
try {
        // On écoute sur le port <PORT>
        ecoute=new ServerSocket(PORT);
        while (true) {
                // On accepte une demande de connexion d'un client
                Socket client=ecoute.accept();
                InputStream in = s.getInputStream();
                OutputStream out = s.getOutputStream();
                ObjectInputStream objIn = new ObjectInputStream(in);
                ObjectOutputStream objOut = new ObjectOutputStream(out);
                Integer I= new Ineger(3);
                objOut.writeObject(I);
                UnObjet O=  (UnObjet)objIn.readObject(O);
                client.close();
        }
}
 catch (IOException e) {
    System.err.println(e.getMessage());
    System.exit(1);
  }
```

Côté client

```
Socket s=null;
  try { s=new Socket(serverhost,PORT); // Création du socket
    // Récupération des flux d'entrée/sortie
     InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();
    ObjectInputStream objIn = new ObjectInputStream(in);
    ObjectOutputStream objOut = new ObjectOutputStream(out);
    Integer I= (Integer)objIn.readObject();
    UnObjet O= new UnObjet() ;
    objOut.writeObject(O);
    s.close();
}  catch (IOException e) {System.err.println(e);}
```

Les bases de données

Pour se connecter à une base de données on utilise l'API JDBC.

Chaque base de données utilise des drivers JDBC différents.

Dans le cas de postgres, on utilise par exemple les drivers «postgresql-8.3-603.jdbc4.jar » .

Pour charger les drivers

```
try {
        Class.forName("org.postgresql.Driver”);
} catch (ClassNotFoundException cnfe) {cnfe.printStackTrace();  System.exit(1);  }
```

Pour se connecter à la base de données

```
try { Connection db = DriverManager.getConnection(url,username,password);
        dbmd = db.getMetaData();
        System.out.println("Connection to "+dbmd.getDatabaseProductName()+" "+
                        dbmd.getDatabaseProductVersion()+" successful.\n");
        ….
        // Fermeture de la connexion vers la base de donnée
        db.close();
 } catch (SQLException se) { se.printStackTrace();  System.exit(1); }
```

Pour faire une requête

```
Try {   Statement st = db.createStatement();
        ResultSet rs;
        rs = st.executeQuery("select …");
        while (rs.next()) {
                rs.getInt(1) ;
                rs.getString(2) ;
        }
} catch (SQLException se) { se.printStackTrace();  System.exit(1); }
                                                        // Retourne la valeur de la deuxième colonne si celle-ci est une chaine de caractère
rs.getString(i) : retourne la valeur de la ième colonne si celle-ci est une chaine de caractère
rs.getInt(i) : retourne la valeur de la ième colonne si celle-ci est un entier
```

Les threads

Un thread est un processus léger, c'est-à-dire qu'il partage la mémoire du processus l'ayant créé.

C'est un ensemble d'instructions s'exécutant en parallèle avec d'autres threads.

Un thread a quatre états (Thread.getState())

1. Etat nouveau : thread déclaré et instancié mais inactif
2. Etat exécutable : à partir du moment où on appelle la méthode start(). Cette méthode appelle après certaines initialisations la méthode run()
3. Etat attente : lorsque le système exécute un autre thread (temps partagé et non parallélisme réel). Il ne consomme pas de ressources. Il est en attente d'une ressource (flux, wait(), …)
4. Etat mort : lorsque le thread est sorti de la méthode run()

L'interface Runnable (java.lang)

Met à disposition la méthode public void run() ;

Le code du thread doit se trouver dans cette méthode.

La classe thread (java.lang)

C'est la classe dont doit dériver notre application pour qu'elle puisse être considérée comme un processus.

```
Public class MonThread extends Thread {…}
MonThread t = new MonThread();
t.start();
```

La classe ThreadGroup

Elle permet de regrouper des threads dans un groupe et d'appliquer des traitements à tous les threads en même temps (destr oy(), isAlive(), …).

Les méthodes de la classe Thread

sleep() : met en pause le thread pendant un certain nombre de millisecondes,
interrupt(), isAlive() : renvoie vraie si le thread est en vie, join() : attente que ce thread meurt,
static Thread currentThread() : permet d'obtenir le Thread courant.

La synchronisation des threads

Lorsque plusieurs threads vont devoir manipuler les mêmes données, il va falloir les synchroniser pour éviter les incohérences.

Il existe plusieurs moyens de synchroniser des threads, certains sont issus des techniques classiques de synchronisation, d'autres sont typiques de java.

1. **Exclusion mutuelle de sections critiques : les verrous**

Pour mettre en place l'exclusion mutuelle, il faut utiliser des **verrous**. Lorsqu'un thread entre dans une section critique, il demande le verrou. S'il l'obtient, il peut alors exécuter le code. S'il ne l'obtient pas, parce qu'un autre thread l'a déjà pris, il est alors bloqué en attendant de l'obtenir.
Une section critique est vue comme une opération **atomique** *(une seule opération indivisible)* par une autre section critique utilisant le même verrou.

a) En java, on utilise le mot clé **synchronized** qui indique qu'une partie du code ne peut être exécuté que par un seul thread à la fois.

- Sur un objet, cela donne le code suivant : synchronized(unObjet) { //section critique }

Il vaut mieux utiliser des références déclarées **final** pour éviter toute modification du verrou.

- Sur une méthode, on a le code suivant : synchronized void methode() { //section critique }

L'usage abusif des méthodes synchronisées peut réduire les performances du programme.

b) Un autre moyen de mettre en place des verrous existe depuis java 1.5. C'est une solution plus complexe mais plus puissante. On utilise le paquetage **java.util.concurrent.locks.** Le code devient :

```
Lock l = new ReentrantLock();
l.lock();
try {
    //section critique
} finally {
    l.unlock();
}
```

l.lock() et **l.unlock()** correspondent respectivement au début et à la fin du bloc **synchronized.**

Cette solution est plus puissante car on peut utiliser différents verrous (Lock) et ainsi être plus précis et donc faire des exclusions mutuelles précises.

2. Synchronisation coopérative

Ici nous allons synchroniser le fonctionnement de plusieurs threads.

1. La méthode la plus simple est d'utiliser les méthodes **wait()** et **notify()** (et éventuellement **notifyAll()**) définies dans la classe **Object**.

- La méthode wait() met en pause le processus appelant.
- La méthode notify() réveille un processus (quelconque) en pause.
- La méthode notifyAll() réveille tous les processus en pause.

Ces méthodes sont applicables sur un objet quelconque mais les notify() ne peuvent réveiller que les processus bloqués sur un wait() sur le même objet.

wait() libère le verrous (celui bloqué par synchronized) lorsque la méthode est appelée si le wait est effectué sur le même objet que le verrou (this.wait()).

Exemple :

```
class ListeTab {
   private String[] tab = new String[50];
   private int index = 0;

   synchronized void ajoute(String s) {
      tab[index] = s;
      index++;
      notify();
      System.out.println("notify() exécuté");
   }

   synchronized String getPremierElementBloquant() {
      //tant que la liste est vide
      while(index == 0) {
         try {
            //attente passive
            wait();
         } catch(InterruptedException ie) {
            ie.printStackTrace();
         }
      }
      return tab[0];
   }
}
```

Cette première méthode permet de traiter les problèmes de synchronisation les plus simples.

2. Le Package **java.util.concurrent.locks**

Ce paquetage permet de résoudre un certain nombre de problèmes des méthodes wait() et notify() notamment lorsque plusieurs ressources (verrous) peuvent être demandées par un seul processus. On utilise ici un seul verrou mais plusieurs variables de condition.

```
private final Lock lock = new ReentrantLock();
private final Condition cond1 = lock.newCondition();
private final Condition cond2 = lock.newCondition();

void methode1() throws InterruptedException {
       lock.lock();
       try {
              cond1.await();
              cond2.signal();
       } finally {
              lock.unlock();
       }
}
```

Les méthodes await() et signal() sont équivalentes à wait() et notify() mais fonctionnent sur des conditions.

Remarques : l'interface **ReadWriteLock** est une interface permettant de résoudre le problème de lecture-écriture vers une variable avec un ou plusieurs lecteurs et un ou plusieurs rédacteurs. Elle est disponible dans le même paquetage.

3. Les sémaphores

Ils sont apparus en java à partir de la version 1.5. De manière classique les sémaphores sont très utilisés mais assez peu en java.

Un **sémaphore** encapsule un entier, avec une contrainte de positivité, et deux opérations atomiques d'incrémentation et de décrémentation.

- variable entière (toujours positive ou nulle).
- opération P (**acquire()**) : décrémente le compteur s'il est strictement positif ; bloque s'il est nul en attendant de pouvoir le décrémenter.
- opération V (**release()**) : incrémente le compteur.

Exemple de semaphore à un jeton :

```
Semaphore sem = new Semaphore(1);
try {
 sem.acquire();
  //section critique
  sem.release();
} catch(InterruptedException e) {
   e.printStackTrace();
 }
```

Le sémaphore diffère du verrou car lorsque l'on fait plusieurs release(), ils sont gardés en mémoire alors que pour un verrou deux déblocages sont équivalent à un seul.

En java 1.5, plusieurs classes sont proposées et utilisent ces outils. On trouve par exemple : **BlockingQueue** (liste bloquante non bornée) et **ArrayBlockingQueue** (liste bloquante à tampon borné).

Un autre outil très utile est **Executor**. Il s'agit d'une liste d'attente bloquante d'actions à effectuer. On retrouve exactement le même mécanisme lorsque l'on utilise le thread dédié à l'affichage graphique *(EventDispatchThread)* de *Swin.*

Exemple :

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {
        public void run() {
        System.out.println("appel asynchrone 1");
        }
});
executor.execute(new Runnable() {
        public void run() {
        System.out.println("appel asynchrone 2");
        }
});
```

L'interblocage

Il apparait lorsque deux threads sont en attente l'un de l'autre

Exemple :

```
class DeadLock {
   private final Object lock1 = new Object();
   private final Object lock2 = new Object();

   void a() throws InterruptedException {
      lock1.wait();
      lock2.notify();
   }

   void b() throws InterruptedException {
      lock2.wait();
      lock1.notify();
   }
}
```

Si un thread execute a() pendant qu'un autre exécute b() alors il y a interblocage.

Il faut éviter lorsque l'on propose des solutions de synchronisation qu'il y ait des interblocages.

La famine

Elle apparait lorsqu'un thread peut se retrouver à n'avoir jamais l'accès au processeur à cause de l'activité des autres threads. Il faut également veiller à ce que cela n'arrive pas dans les solutions de synchronisation proposées.

Remarque

La synchronisation coûte cher, il ne faut pas en abuser.

Les interfaces graphiques

Il existe deux API java pour développer des interfaces graphiques (IHM ou GUI). La plus ancienne est AWT, la plus récente SWING. Swing ne remplace pas AWT puisqu'il utilise certaine de ses fonctionnalités mais il remplace les composants par des composants 100% java. Une autre importante différence est la gestion des évènements (interactions avec les utilisateurs).

La partie visuelle des interfaces est basée sur le modèle suivant : les fenêtres (Frame, JFrame, …) sont représentées par des conteneurs (Container), qui regroupent les différents composants (Component) tels que les boutons (Button, JButton), les zones de saisie (TexteField, JTexteField), …..

Awt

Vous trouverez ici quelques informations sur AWT sans détails, l'API Swing étant l'API que nous allons utiliser.

Les classes nécessaires sont disponibles dans les API java.awt et java.awt.event.

La classe **Component** est la base des objets graphiques, elle regroupe des méthodes (setVisible() pour rendre visible l'élément ou setBounds() pour définir la taille et la position de l'élément par exemple), utiles à tous les objets graphiques.

Parmis les composants on trouve par exemple : Label, Button, TextField, …

La classe **Container** permet de définir un objet graphique capable de contenir d'autres objets graphiques. Elle contient notamment la méthode add() permettant d'ajouter des objets au conteneur.

La classe **Window** est un type particulier de conteneur ayant déjà une bordure, une barre de titre, des boutons de minimisation, maximisation et fermeture et qui peuvent être repositionnées et redimensionnées par l'utilisateur.

La classe **Frame** est dérivée de Window et sera la classe de base pour les fenêtres graphiques.

Swing

Les classes nécessaires sont disponibles dans l'API javax.swing mais également dans les API AWT.

Les conteneurs

Ils sont destinés à contenir des composants.

La classe **JWindows** est la classe permettant de représenter une fenêtre de manière simple.

La classe **JFrame** est la version Swing de la classe Frame, la plus part des applications swing utilisent au moins une JFrame. C'est le premier élément à créer. Elle comporte beaucoup de méthodes permettant de gérer la fenêtre (add() : ajout de composants, getJMenuBar() : récupération du menu, …).

On peut dériver la classe JFrame pour créer son propre type de fenêtre.

Exemple :
```
JFrame frame = new JFrame("Ma Frame") ;
frame.setSize(200,300) ;
frame.setVisible(true) ;
```

La classe **JDialog** c'est une boîte de dialogue qui permet d'avoir une fenêtre avec la notion de modalité (seule cette fenêtre est active).

Les composants

La classe **JComponent** est la version swing de la classe Component qui est la base de tous les composants Swing. Voici une liste non exhaustive de composants disponibles (il en existe plus de 40) : JButton, JLabel, JTextArea, JTextField, JEditorPane, JRadioButton, JCheckBox, ButtonGroup, JPasswordField, JComboBox, JList, JScrollBar, JToolBar, JTree, JTable, JSlider, JFileChooser, JColorChooser, JProgressBar, JMenu, JPopupMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem, JToolTip, JSeparator, JTabbedPane.
Chaque composant possède des méthodes permettant de définir son aspect graphique (setText() par exemple pour JLabel, addItem() pour un JList).
Pour ajouter un composant, on l'ajoute au container d'une fenêtre que l'on récupère avec la méthode getContentPane().

Exemple : frame.getContentPane().add(new JLabel(« Bonjour »)) ;

La classe **JPanel** définie des zones permettant d'organiser et de contrôler la répartition des composants dans une fenêtre. La procédure classique pour ajouter des composants à une fenêtre est de définir un JPanel, d'y ajouter des composants puis d'ajouter le JPanel à un JFrame. On peut passer les composants directement au JFrame mais il est préférable de passer par un JPanel.

Layout

Le layout est l'organisation spatiale des composants dans un Panel ou une fenêtre. C'est le **LayoutManager** qui gère cette organisation et qui place les composants en fonction : de la taille des composants, de leur placement par rapport aux autres composants, de la politique de placement, …

Il existe 7 méthodes de placement différentes. Par défaut, Swing utilise le FlowLayout qui place les composants les uns après les autres par lignes successives en fonction de la largeur de Panel ou de la fenêtre.

Pour changer le Layout, on utilise la méthode **setLayout()**. Si on utilise plusieurs panel dans une même fenêtre, chaque panel peut avoir des layout différents. On peut désactiver le layout manager en passant **null** comme paramètre de setlayout(), dans ce cas, c'est l'utilisateur qui positionne et dimensionne les composants avec la méthode setBounds() propre à chaque composant.

Les différents layout sont : **Flowlayout** (de gauche à droite, de bas en haut), **GridLayout** (on définit une matrice de composants), ViewportLayout (vue partielle des composants), BoxLayout (boîtes horizontales et verticales imbriquées), CardLayout (tas de composants, seul le sommet est visible), OverlayLayout (idem précédent avec gestion des tailles différentes), **BorderLayout** (faon géographique : nord, sud, est, ouest, centre), **GridBagLayout** (basé sur des contraintes entre composants).

Contexte graphique et méthodes d'affichage

Pour pouvoir afficher quelque chose, un composant doit posséder un contexte graphique, celui-ci est passé à chaque appel des méthodes paint(), paintComponent(), paintBorder() ou paintChildren() qui permettent l'affichage. C'est dans ces méthodes que l'on peut gérer précisément l'affichage, par exemple en dérivant d'un composant ou d'une fenêtre et en les redéfinissant.

Pour récupérer le contexte graphique d'un élément, et ainsi modifier son affichage depuis l'extérieur du composant, il faut faire appel à la méthode **getGraphics()** qui renvoie un objet de type **Graphics.**

Un certain nombre de méthodes de dessin sont disponible dans la classe Graphics. Par exemple : drawLine(x1, y1, x2, y2) qui permet de tracer une ligne, drawPolygon(), drawRect(), drawString(), fillRect(), copyArea(), drawImage(), …

Les fonds graphiques : la classe Canvas

La classe Canvas est une classe générique qui doit être dérivée pour être utilisée. Elle sert à créer de nouveaux composants.

Les éléments décoratifs

Swing met à disposition la classe **ImageIcon** et les classes dérivées de **AbstractBorder.** Un objet de type ImageIcon permet de charger une image et de l'utiliser comme fond dans un composant. Il

existe 8 types de cadres (Border) définis : BevelBorder (cadre 3D), CompoundBorder, EmptyBorder, EtchedBorder, LineBorder, MatteBorder, SoftBevelBorder, TitledBorder.

Un cadre est créé à l'aide d'un objet de type BorderFactory (Border cadre=BorderFactory.createLineBorder(Color.black)) et associé à un composant grâce à la méthode setBorder() (JButton b=new JButton() ; b.setBorder(cadre) ;).

La gestion des interactions

Après avoir défini tous les éléments graphiques précédents, on obtient l'affichage statique de notre interface graphique. Mais celle-ci n'est pas fonctionnelle (à quelques exceptions près comme par exemple la possibilité d'activer le bouton de fermeture de l'application). Pour définir le comportement de l'interface en fonction des interactions avec l'utilisateur, il faut s'intéresser à la gestion des évènements en Swing.

Les évènements

Le gestionnaire d'évènements permet d'intercepter les actions des utilisateurs et d'assigner au programme un comportement adapté en réponse. Il existe de nombreuses manières de gérer les évènements, nous en verrons quelques exemples.

Chaque composant peut recevoir un certain nombre d'évènements. Ces évènements sont hérités de la classe **java.util.EventObject**. Pour réagir à ces évènements, il est nécessaire de définir des écouteurs (Listener) qui vont écouter certains évènements et exécuter une action lorsqu'un évènement se produit. Les écouteurs doivent hériter de **java.util.EventListener**.

On doit donc définir pour un composant l'ensemble des évènements auxquels il réagit en lui associant des écouteurs sinon les évènements n'auront aucun effet sur le composant.

Les évènements sont générés par l'utilisateur et envoyés vers les composants ou conteneurs. Par abus de langage on dit que ces objets génèrent les évènements directement.

Les catégories d'évènements disponibles en Java sont listées ci-après, chaque catégorie donne lieu à un évènement en ajoutant **Event** au nom de la catégorie.

Les évènements de base :

1. **Mouse** : associé à la souris (click, bouton enfoncé, bouton relâché, souris entrée ou sortie du composant, …
2. **MouseMotion** : relatif au mouvement de la souris : move, drag, …
3. **MouseWheel** : relatif à la rotation de la roue de la souris
4. **Key** : associé au clavier : touche tapée, enfoncée ou relâchée.
5. **Component** : changement de la visibilité, de la position ou de la taille d'un composant.
6. **Container** : un composant a été ajouté ou retiré d'un container.
7. **Focus** : gain ou perte de focus
8. **Window** : modification du statut de la fenêtre (activée, iconifiée, …).

Les évènements spécifiques à certains composants :

1. **Action** : généré par : un bouton actionné (JButton, JRadioButton, JCheckBox, …), un élement sélectionné dans un JComboBox, la touche Entrée appuyée dans un JTextFiel, un fichier sélectionné dans un JFileCHooser, un élément sélectionné dans un JMenu, JMenuItem, …
2. **Adjustment** : nouvelle valeur sélectionnée dans un JScrollBar
3. **Caret** : modification de la position du curseur dans un JTextComponent
4. **Change** : changement d'état relatif à la plus part des composants.
5. **Document** : modification des attributs d'un document (insertion de texte, ….)
6. **Hyperlink** : activation d'un hyperlien
7. **InternalFrame** : modification du statut d'une fenêtre pour le composant JInternalFrame
8. **Item** : sélection d'une case à cocher.
9. **ListData** : modification du contenu d'une JList.
10. **ListSelection** : sélection dans une JList
11. **MenuDragMouse** : souris draguée, entrée ou sortie d'un JMenuItem
12. **Menu** : sélection dans un JMenu
13. **MenuKey** : touche de menu préssée
14. **PopupMenu** : sélection dans un menu popup
15. **TableColumnModel** : concerne les JTable
16. **TableModel** : concerne les JTable
17. **TreeExpansion** : déploiement ou repli d'un JTree
18. **TreeModel** : modification, insertion ou suppression d'un noeud d'un JTree
19. **UndoableEdit** : occurrence d'un undo dans un JTextComponent

Un évènement est un objet, il possède donc des méthodes. La méthode **getSource()**, par exemple, renvoie la référence du composant ayant généré l'évènement. Pour un évènement de type MouseEvent, **getX(), getY()** renvoient les coordonnées de la souris au moment de l'évènement.

Les écouteurs

Pour qu'un composant réagisse à un évènement, il faut lui ajouter un écouteur pour cet évènement (à condition que le composant puisse recevoir ce type d'évènement). On procède de la manière suivante : leComposant.add**Categorie**Listener(un**Categorie**Listener) ;
En remplaçant Categorie par une des catégories précédemment définies.
Exemple : JButton b = new JButton(« executer"); b.addActionListener(new ActionListener() {…});

Remarques : un évènement peut être écouté par plusieurs écouteurs. Un écouteur peut écouter plusieurs évènements en même temps.

Le code à exécuter lors de la réception d'un évènement doit être mis dans certaines méthodes des écouteurs. Chaque méthode porte un nom particulier, mais toute prennent en paramètre un objet dérivé du type Event en fonction de l'évènement généré. Par exemple, pour un évènement de type **ActionEvent**, on appellera la méthode **addActionListener()** en lui passant un **actionListener** dont on aura codé la méthode **actionPerformed()** prenant en paramètre un objet de type **ActionEvent**.

Les écouteurs doivent être absolument implémentés car il s'agit d'interfaces. Pour cela, il existe plusieurs façons de procéder.

1. On déclare une classe qui implémente un des écouteurs.
   Exemple : public class EcouteurBouton implements ActionListener {

```
        Public void actionPerformed(ActionEvent e) { System.out.println("Bouton préssé");
}
}
Public class maFame extends JFrame {
        public maFrame() {
                JButton b = new JButton("Mon bouton");
                b.addActionListener(new EcouteurBouton());

        }

}
```

L'avantage de cette solution est de pouvoir réutiliser l'écouteur pour un autre bouton. On peut par ailleurs déclarer cette classe en classe imbriquée, c'est-à-dire déclarée et visible uniquement à l'intérieur d'une classe.

2. La fenêtre est l'écouteur des composants qui lui sont rattachés.
   Exemple : public class maFrame extends JFrame implements ActionListener {
   public void actionPerformed(ActionEvent e) { System.out.println("Bouton préssé"); }
   public maFrame() {
   JButton b = new JButton("Mon bouton");
   b.addActionListener(this);

   }

   }

3. Les classes anonymes : on peut implémenter directement les interfaces au moment de leur utilisation. Ceci est plus rapide mais empêche la réutilisation des écouteurs et ne doit pas être utilisé si le code des méthodes est trop long.
   Exemple : public maFrame() {
   JButton b = new JButton("Mon bouton");
   b.addActionListener(new ActionListener() { public void actionPerformed() { System.out.println("Bouton préssé"); } });

   }

Remarque

Il existe également des classes d'adaptation qui permettent de ne pas coder toutes les méthodes de l'interface si on ne s'intéresse qu'à une seule d'entre elles. Ces classes sont construites de la manière suivante : CategorieAdapter (Exemple : ActionAdapter). Elles implémentent toutes les méthodes de l'interface avec un code vide.

Gestion des images

Java permet de gérer les images telles que gif, jpeg et png.

ImageIcon

La manière la plus simple de gérer les images est d'utiliser la classe **ImageIcon**.

Exemple : ImageIcon Im = new ImageIcon(« image.gif ») ;

La classe ImageIcon propose également une méthode renvoyant l'image associée (**Image getImage()**).

Les ImageIcon peuvent être utilisées dans les constructeurs des composants Swing.

Exemple : JLabel label = new JLabel(Im) ;

Image

SI on veut gérer l'affichage de l'image grâce au contexte graphique, il faut passer par un objet detype **Image**.

La classe Image propose une série de méthode permettant par exemple de retailler l'image et d'en obtenir une copie (getScaledInstance()).

Java met également à disposition la classe ToolKit pour manipuler les images.

Exemple : Toolkit tool = Toolkit.getDefaultToolkit();

Image Im = tool.getImage("image.gif");

La classe Graphics avec la méthode **drawImage()** permet d'afficher une image. Le contexte graphique est accessible notamment dans les méthodes paint() des composants ou fenêtres swing. Cette méthode demande également de définir un **ImageObserver** qui est une référence vers l'objet écoutant les demandes de mise à jour de l'image. Le JPanel dans lequel sera affichée l'image peut jouer ce rôle.

Les Animations

Pour réaliser une animation, on code une boucle dans le run() d'un thread dans lequel on appelle des repaint() en faisant éventuellement des sleep() pour temporiser.

Il faut par ailleurs se soucier de quelques détails pour que l'animation soit fluide comme le double buffer, …

Les applets

Les applets sont des applications destinées à s'exécuter dans un navigateur.

Au lieu d'étendre la classe JFrame, on étend la classe JApplet qui dérive de Panel. Une applet a donc accès à l'affichage graphique par l'intermédiaire de la méthode paint() (cf Swing).

Au lieu de déclarer une méthode main() on déclare une méthode init().

A chaque démarrage d'une applet, la méthode start() est appelée. Deux autres méthodes (stop() et destroy()) sont également disponibles dans la classe applet.

Exemple

```
//importation des différentes bibliothèques
import java.applet.*;
public class Hello extends JApplet{
   public void init() {  //code d'initialistaion    }
   public void start(){  //code de d'éxécution   }
   public void stop() {  //code de suspension de l'execution   }
   public void destroy() { //code de terminaison  } }
```

Pour intégrer une applet dans une page HTML on procède de la manière suivante

```
<APPLET code="monApplet.class" width="500" height="200">
Applet java
</APPLET>
```

La réflexivité

Il est possible de récupérer des informations sur les objets de manière dynamique comme par exemple, le nom de la classe, le nom et les paramètres des méthodes, … Ce procéder porte le nom de réflexivité en java et est rendu possible grâce aux classes suivantes : **Java.lang.Class**, **java.lang.reflect.Field**, **java.lang.reflect .Method, Java.lang.Package.**

Chaque classe possède une méthode **getClass()** (provient de la classe Object) qui renvoie un objet de type **Class.**

On trouve dans la classe Class entre autre les méthodes suivantes : getConstructors(), getField(), getMethods(), getSuperclass(), getPackage(), … Ces méthodes permettent d'obtenir les informations sur les méthodes et attributs d'une classe.

Exemple : Point p1 = new Point(3, 4); System.out.println("Class Name:" + p1.getClass().getName());

Affiche : Class Name : Point

Pour récupérer la signature d'une méthode, on utilise les méthodes suivantes :

```
Class c = Class.forName("maClasse");
java.lang.reflect.Method[] m = c.getMethods();
```

```
 Class r = m[0].getReturnType();
Class[] params = m[0].getParameterTypes();
```

Les méthodes **set()** et **get()** permettent également de récupérer ou modifier la valeur d'un champ(cf classe Field).
De meme la method **invoke()** permet d'exécuter dynamiquement une méthode (cf classe Method).

Les annotations

Depuis Java 1.5, il existe la possibilité de déclarer des annotations. Ces annotations permettent de renseigner des parties du code (méthodes, attributs,…) comme dans le cas de javadoc mais avec pour objectif de donner des informations lors de la compilation ou l'exécution du programme.

On note une annotation de la manière suivante

Déclaration : Public @interface monAnnotation { …}

Utilisation : @monAnnotation public Type unAttribut ;

Exemple :

```
public @interface SimpleAnnotation { }

public @interface AttributAnnotation { String value(); int count(); }

@SimpleAnnotation
public class MaClasse {

@SimpleAnnotation
protected String name;

@SimpleAnnotation protected int value;

@AttributAnnotation(value="info", count=3);
public MaClasse () { }

@SimpleAnnotation
@AttributAnnotation(value="m", count=1);
public void method () { }
```

Les types énumérés

Introduit avec java 1.5, les types énumérés permettent de définir des types ayant un ensemble limité de valeurs possibles.

Dans sa forme la plus basique, une **enum** contient simplement la liste des valeurs possibles qu'elle peut prendre. Elle se déclare comme une classe mis à part que l'on utilise le nouveau mot-clef **enum**, par exemple :

```
public enum Saison { printemps, ete, automne, hiver; }
```

Les enum sont des classes, on peut donc ajouter des attributs et des méthodes (dont des constructeurs).
Chaque enum possède deux méthode **values()** qui renvoie la liste des valeurs possibles et **valueOf()** qui renvoie une valeur précise. En effet on peut associer une valeur à chaque élément de l'énumération (exemple : ete(« summer »))

On ne peut pas hériter d'un enum et on ne peut pas instancier un enum (pas de new), le constructeur est uniquement là pour l'initialisation.

Gestion des sons

Java permet de lire des sons de différents formats (wav, midi, …).

La classe utilisée pour lire les sons doit implémenter l'interface **AudioClip** qui possède les méthodes **play(), stop(), loop().**

On peut jouer de sons de plusieurs manières.

La façon la plu simple est d'utiliser l'API Applet.

Exemple :
URL url = this.class.getResource(« hit.wav");
AudioClip clip = Applet.newAudioClip(url);
clip.play();

Il existe d'autres façons de procéder, non décrites ici, qui permettent une gestion plus complète des sons. L'API de référence est : javax.sound

Les arguments variables

Cette nouvelle fonctionnalité de java 1.5 permet de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci évite de devoir encapsuler ces données dans une collection. La notation utilise trois petits points : ...

```
public class TestVarargs {
  public static void main(String[] args) {
    System.out.println("valeur 1 = " + additionner(1,2,3));
    System.out.println("valeur 2 = " + additionner(2,5,6,8,10));
  }
  public static int additionner(int ... valeurs) {
        int total = 0;
        for (int val : valeurs) {
                total += val;   }
        return total;
  } }
```

Evolution du langage avec Java 1.5

Import static : permet d'importer des classes statiques et d'utiliser les attributs statiques sans préfixer par le nom de la classe.

Au lieu de

```
public class TestStaticImportOld {
  public static void main(String[] args) {
    System.out.println(Math.PI);
    System.out.println(Math.sin(0));  }
}
```

On a

```
import static java.lang.Math.*;
public class TestStaticImport {
  public static void main(String[] args) {
    System.out.println(PI);
    System.out.println(sin(0));  }}
```

L'auto-boxing

Les types primitifs sont convertis directement lors de leur affectation dans un objet.

Au lieu de

```
import java.util.*;

public class TestAutoboxingOld {
  public static void main(String[] args) {
    List liste = new ArrayList();
    Integer valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = new Integer(i);
      liste.add(valeur);    }   }     }
```

On a

```
import java.util.*;

public class TestAutoboxing {

  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }
  }

}
```

Les extensions de java

On trouve de nombreuses extensions java sur le site de sun ou ailleurs. Par exemple, java2D et java3D pour la gestion des affichages 2D et 3D.

Optimisation de Java

Compilez optimisé

Utiliser le paramètre -O lors de la compilation. Le compilateur est alors capable de placer en ligne les méthodes private et final.

Déclarez des méthodes statiques

Cela améliore la vitesse d'exécution.

Réutilisez les instances

Créer des objets est plus coûteux que les réutiliser. Déclarez des instances statiques et réinitialisez-les.

Divers

La classe PasswordAuthentification : contient un userName et un Password.

System.currentTimeMillis()