



***Votre avenir
nous engage***

Découverte de la programmation objet en

Java

Support de Formation

Voici la liste des documents nécessaires pour suivre ce module de formation :

Apprentissage d'un langage de programmation orienté objet : Java

Auteur : André-Pierre Limouzin, fichiers java_asf.pdf, java_bsf.pdf, java_csf.pdf.

Ces trois documents sont un cours d'autoformation à la programmation objet avec corrigés.

Programmer en java

Auteur Claude Delannoy (éditions Eyrolles).

Un livre expliquant les principes de la programmation objet avec java.

Swing, la synthèse

Auteurs Valérie Berthié et Jean-Baptiste Briaud (éditions Dunod).

Un livre donnant des exemples d'utilisation de classes d'interface Java.

Ces deux livres ne sont cités qu'à titre d'exemple, d'autres livres du commerce peuvent vous rendre le même service, et je pense qu'il est utile de se déplacer dans une librairie, ou une bibliothèque et de choisir les livres qui vous conviennent.

Certains exercices font référence aux polycopiés de André-Pierre Limouzin (dans la rubrique conseil). Dans tous les cas, il est recommandé de faire des recherches bibliographiques sur chaque sujet traité dans ce support, car ce document est un guide de travail, mais pas un cours.

Introduction

La formation à un nouvel outil, ou à une nouvelle démarche, s'appuie souvent sur la formation et l'autoformation. La première permet d'avoir un guide, une philosophie, et un garde fou. La deuxième va vous permettre d'avoir accès à une multitude de documents qui vont vous permettre de vous enrichir, à condition d'avoir les clefs de lecture de ces documents.

De manière générale, vous trouverez, dans ces documents, trois ingrédients qui feront de vous des programmeurs java :

- De la méthode (dans les discours et dans les exemples). Vous en trouverez très peu, même si la méthode commence à poindre son nez dans les derniers livres java parus.
- De la technique sur la programmation objet (encapsulation, héritage, polymorphisme). Ici vous en trouverez beaucoup plus, même si les exemples traités ne suivent pas toujours les belles intentions affichées.
- De la connaissance sur les classes java et leur utilisation. Vous en trouverez beaucoup, les sujets sont rarement traités de manière exhaustive, mais fournissent de très bonnes introductions, voire des exemples complets, qui permettent de maîtriser les classes java, dans la mesure où vous prendrez le temps d'une synthèse sur le sujet, et que vous réécriviez les exemples en respectant les standards du développement objet qui sont, eux, bien souvent maltraités.

Le but de ce document est de vous permettre d'acquérir suffisamment de méthode, et la technique objet nécessaire pour comprendre tous les codes java proposés dans la littérature, et sur le net, de reformuler ces codes pour les transcrire dans les standards du développement objet (c'est important car les nouvelles acquisitions de connaissances se font souvent à partir d'exemples). Ce document vous permettra aussi de concevoir et de coder vos classes pour qu'elles soient conformes aux règles de développement objet. Ainsi, vos applications seront plus stables, et évolueront plus facilement. Les connaissances des classes java qui seront abordées au cours des exercices, ne sont pas ici une priorité : c'est l'aspect que l'on appréhende le plus facilement en autoformation, et avec l'expérience, à condition d'avoir les bases que nous allons vous apporter.

Les exercices ont d'abord pour but de mettre en place les techniques objet, les connaissances venant petit à petit suivant les besoins. Puis, au fil des exercices, nous rencontrerons les problèmes de conception. Nous résoudrons ces problèmes par le bon sens, en essayant de mettre en place des règles méthodologiques qui nous conduisent à une bonne pratique.

Le module de formation « Une démarche d'analyse et de conception avec UML » viendra apporter le complément théorique sur ces règles méthodologiques, montrant d'ailleurs que la bonne pratique, éprouvée sur le terrain, a permis de construire une méthode nous faisant profiter de l'expérience des pionniers de la programmation objet. Donc, pas de gourou, pas de technocrate qui dicte du haut de sa chaire le bien et le mal (objet), mais une pratique du terrain sur laquelle un recul a été pris pour en tirer la substantifique moelle, et vous en faire profiter.

Présentation du document

Ce document a été découpé en 6 parties, dont les trois dernières peuvent être traitées indépendamment.

1) Découverte des principes de la programmation objet.

Ici seront abordés les principes de protection des données (l'encapsulation), les interfaces des objets (les protocoles), les évolutions des objets dans le temps (l'héritage) et de leurs comportements (polymorphisme).

2) Réalisation d'interfaces avec AWT

Ici, nous commencerons par étudier les applets, puis le traitement des événements en java, puis nous réaliserons des applications avec IHM. Nous serons alors amenés à nous poser un certain nombre de questions de méthode de conception.

3) Rendre un objet métier écoutable

Nous allons appliquer les techniques de programmation événementielle aux objets métiers. Mon programme est averti qu'un objet a changé d'état, comme il est averti qu'un bouton a été appuyé. Ces techniques sont indispensables pour programmer plusieurs vues sur un même « document », et que ces vues soient synchronisées.

4) Accès aux bases de données avec JDBC

Ici nous examinerons les différentes possibilités d'accéder à des bases de données par JDBC, via ODBC : requêtes immédiates, requêtes pré compilées (ou paramétrées), appel de procédures stockées (avec paramètres en entrée, en sortie, en entrée/sortie, avec résultat, avec valeur de retour), et sécurisation des transactions.

5) Threads et sockets

Ici nous étudierons les échanges d'informations entre machines distantes à l'aide des sockets, puis nous étudierons les processus légers (ou threads), afin de réaliser un serveur multi clients.

6) Réalisation des interfaces java avec les classes Swing

Ici nous verrons les principes de fonctionnement des classes SWING, puis nous aurons un aperçu des outils à mettre en œuvre pour réaliser des interfaces professionnelles.

Pré requis pour suivre ce module :

La connaissance de l'algorithmie et de la programmation procédurale est indispensable pour démarrer cette formation. Ces deux formations sont disponibles sous le même format.

La partie base de données pré suppose la connaissance de la programmation des bases de données : SQL, procédures stockées, transactions, ... Cette formation est également disponible sous le même format.

Première partie : Découverte des principes de la programmation objet

Dans cette première partie nous allons découvrir les principes de la programmation objet. Pour cela nous allons définir une classe (comme nous avons définis un enregistrement en algorithmie), puis, petit à petit, en appliquant les principes de la programmation objet, nous allons en faire une vraie classe.

Nous allons ensuite bâtir des classes, à partir de cette première classe, (par héritage), et nous allons implémenter des comportements mutants (par polymorphisme).

Quand nous aurons fini ce tour préliminaire, nous reprendrons les programmes de programmation procédurale de la pile et de la table de noms alphabétique, pour les transformer en classes.

Enfin nous ferons un travail de synthèse sur un exemple pris de bout en bout, mais sur lequel vous ne serez plus guidés. Il vous faudra alors prendre des décisions de conception (pour la définition des interfaces de vos classes) . Cet exemple vous permettra de découvrir la notion de collection (*) en java.

(*) Une collection est une structure de données qui permet de conserver un certain nombre d'informations de manière structurée et organisée. Ces collections sont des classes java qu'il ne faut surtout pas réinventer, quand nous aurons appris à les utiliser. Exemples de collections : un ensemble, un tableau dynamique, un dictionnaire, une arborescence, une liste, une liste chaînée, ...

Exercice 1 : de l'enregistrement à la classe Java

Nous désirons gérer les informations concernant des salariés d'une entreprise.

- Plutôt que de ranger les nom, prénom, adresse de chaque salarié dans des variables isolées, il est préférable de créer un type composé qui représente la fiche salarié, semblable à l'enregistrement en algorithmique.
- Créez une classe Individu qui regroupe ces trois informations : dans cette version, les données membres seront déclarées **public**, pour que le main puisse les utiliser de l'extérieur de la classe (ce qui est tout à fait provisoire et absolument contraire à la méthode objet !)

Conseils :

- Après avoir déclaré une référence sur un objet de la classe Individu (**Individu leMeilleur ;**) n'oubliez pas de créer cet objet par un **new** avant de l'utiliser (**leMeilleur = new Individu () ;**)
- Pour accéder aux données membres à partir du main, utilisez la notation pointée (comme en algo pour accéder aux membres d'un enregistrement) (**leMeilleur.nom**)
- Notez l'erreur de compilation lorsque l'on tente de l'utiliser avant sa création
- Après compilation, vérifiez que vous avez une nouvelle classe (fichier **Individu.class**) dans votre répertoire.

Quand vous développez une application, l'atelier de développement vous signale de temps en temps des erreurs (que ce soit à la compilation, ou à l'exécution). Il est important de bien lire ces messages d'erreur, même s'ils semblent abscons, car ces messages doivent vous permettre de corriger vos erreurs. La technique consistant à faire des corrections aléatoires, technique éprouvée par des générations de développeurs, ne mène pas à grand-chose, si ce n'est à introduire de nouvelles erreurs, et quand bien même l'erreur disparaîtrait, vous n'aurez pas compris votre erreur, donc vous la referez.

Exercice 2 : Protection des informations contenues dans la classe, contre les programmeurs hackers !

Quand un développeur réalise des sous programmes, il le fait dans une certaine logique. L'utilisation de ces sous programmes requiert donc de rentrer dans cette logique pour utiliser ces outils au mieux. L'expérience montre qu'un grand nombre de programmeurs n'utilisent pas ces outils au mieux, et ce pour différentes raisons :

- Mauvaise compréhension des outils.
- Manque de confiance dans les outils.
- Intervention sur le fonctionnement interne des outils.
- Voire réécriture complète des outils...

Dans la programmation objet, un outil sera livré sous forme d'une classe qui explique clairement son utilité, et les services qu'elle propose. Cette documentation au format HTML va permettre aux utilisateurs de comprendre le rôle de l'outil, comment s'en servir, sans savoir comment il est réalisé. Cette documentation est une présentation HTML des commentaires de réalisation de l'outil. Il va sans dire que cette documentation est obligatoire. Elle se réalise au moment où l'on écrit les interfaces des méthodes de la classe, et se valide, avant de valider le code de la classe. Un exemple va vous être présenté ultérieurement.

Les données de notre classe doivent être inaccessibles, et le fonctionnement interne de la classe caché. Ainsi les développeurs qui utilisent nos classes devront faire confiance à nos développements (qui doivent être à la hauteur !!!), et ne se poseront pas de questions idiotes sur ce qui ne les regarde pas.

Quand j'utilise une machine à laver le linge, je m'intéresse aux différents services que peut me rendre l'appareil, ainsi qu'à son mode d'emploi, mais l'électronique, ou la mécanique, qui la fait fonctionner ne m'intéresse absolument pas (tant que la machine fonctionne ...). Pour rien au monde je n'essaierai de fabriquer une machine à laver, si j'en ai une qui fonctionne sous la main (cela tombe sous le sens, mais manifestement pas pour tous les informaticiens ...).

Nous voilà partis dans l'aventure qui consiste à faire de notre enregistrement salarié une classe.

- Actuellement, le programme principal peut ranger n'importe quoi dans nom, prénom, adresse : une chaîne vide, des chiffres ...
- La notion **d'encapsulation** permet à l'objet de rester maître de ses données membres, en vérifiant la cohérence des modifications demandées par le programme principal.
- Dans la classe Individu, déclarez le nom, prénom et adresse en **private**, de façon à ce qu'ils ne soient pas accessibles directement d'une autre classe ou du main.
- Générez le projet et noter l'erreur de syntaxe : le main ne peut plus du tout accéder aux données membres de l'objet.
- Réalisez des fonctions élémentaires **getNom**, **setNom**, **getPrenom**... qui permettent au programme principal d'accéder à tous les champs privés dans l'objet, en respectant son intégrité : dans les fonctions **setNom** et **setPrenom** on interdira les chaînes vides et on n'acceptera que les lettres minuscules et majuscules. On n'effectue pas de contrôle sur l'adresse. Ces méthodes sont définies dans la classe, et ont accès à toutes les informations de la classe (même privées). Les méthodes set sont appelées des modificateurs, et en général rendront un booléen qui précise si le set s'est bien passé. Le mieux est de toujours retourner un booléen dans un set, ainsi vous préparez les évolutions futures de vos contrôles de cohérence. Les méthodes get sont les accesseurs (ou

sélecteurs). Les méthodes get retournent la valeur de l'attribut de la classe concernée. Notons que les get et set respectent le type de l'attribut concerné dans la classe. Vous regarderez un exemple, avec sa documentation générée par l'outil javadoc, fourni en fin de cet exercice.

- Réalisez des fonctions membres **lire**, **afficher** qui permettent à l'objet de saisir ses données membres au clavier et de les afficher à l'écran
- Dans le programme principal, appelez la méthode **lire** de la classe Individu puis **afficher** pour vérifier les données saisies.
- Dans le programme d'essai, proposez à l'utilisateur une modification du nom et du prénom, après la première saisie de l'objet pour tester les fonctions élémentaires **setNom** et **setPrénom**. Réaffichez ensuite l'objet. Vérifiez que ces fonctions contrôlent bien la cohérence du nom et du prénom.
- Réécrire la fonction lire pour y intégrer le contrôle sur le nom et le prénom : la fonction doit relire les champs jusqu'à ce qu'elle obtienne des données correctes. Réalisez cette nouvelle fonctionnalité en écrivant le moins de code possible grâce aux fonctionnalités existantes ! (cette manière de faire doit absolument être mise en place dans tout développement de classe : une couche d'outils qui fait les get et set, tous les autres outils s'appuyant sur cette couche. Ainsi, toute modification de cohérence sur un set par exemple, est répercutée sur l'ensemble des méthodes de la classe qui modifie cet attribut. Rappelez-vous du principe objet : jamais de copier coller, c'est-à-dire jamais de duplication de code) Le programme principal est-il modifié par ce rajout ? (merci l'encapsulation !)

Conseils

- N'oubliez pas l'attribut **public** devant les fonctions membres, sinon votre classe sera tellement bien protégée qu'elle en deviendra inutilisable. Nous verrons plus tard dans quel cas ces fonctions membres deviendront privées.
- Les données membres commenceront toutes par le préfixe **m_typeNom** : m_strNom Ceci est une convention qui permet de mettre en évidence les méthodes qui manipulent les attributs (en général les get et set, mais il y a des exceptions). Les autres variables locales ne seront pas préfixées par m_ qui veut dire donnée **membre**.
- Faire des fonctions booléennes pour **setNom** et **setPrenom** afin d'avertir le programme appelant de l'issue de l'opération. La prudence nous le fera faire également pour la méthode setAdresse, bien qu'il n'y ait pas de contrôle, car une vérification de la cohérence de l'adresse dans une version future n'est pas à écarter. Les méthodes commencent toutes par une minuscule.
- Faire des fonctions qui retournent le type String pour **getNom**, **getPrenom** (Attention : à ce stade, nous n'avons pas encore vu comment utiliser des paramètres en sortie en Java. La seule possibilité est donc d'utiliser une valeur de retour)
- La vérification du nom et du prénom utilisant la même règle, les fonctions **setNom** et **setPrenom** devront appeler une fonction **estAlpha** privée (privée car elle appartient à la tripaille interne de la classe, et personne n'a à la connaître de l'extérieur). Il existe une autre solution pour la méthode estAlpha : cette méthode est générale, et peut servir dans d'autres classes. Il est donc préférable de la mettre dans une classe de type « Boîte à outils », c'est-à-dire une librairie de fonctions de travail. La méthode devient alors publique et statique, dans la boîte à outils. Ayez autant de boîtes à outils que nécessaire. Vous ne rangez pas les ciseaux à bois avec les truelles et les burins. Par contre une méthode **estNuméric** se rangerait dans la même Boîte à outils. Pour réaliser cette méthode, la méthode matches de la classe String peut être utile, éventuellement avec l'expression régulière « `[\\p{L}]` » qui sélectionne les lettres y compris les accents.

- Lisez (Limouzin Java 2 chapitres 1 et 2)
Soit une classe Trajet décrivant le trajet entre Pont de Claix et une autre ville en donnant la distance.

Voici la documentation produite :

Class Trajet

```
java.lang.Object  
└─ Trajet
```

```
public class Trajet  
extends java.lang.Object
```

Cette classe décrit un trajet à partir de Pont de Claix.

Version:

1 Created on 4 janv. 2005

Author:

corre

Constructor Summary

[Trajet](#) ()

Method Summary

<code>int</code>	<code>getDistance()</code> cette méthode permet de récupérer la distance du trajet en kilomètres
<code>java.lang.String</code>	<code>getVille()</code> cette méthode donne la ville concernée par le trajet
<code>boolean</code>	<code>setDistance(int distance)</code> cette méthode met à jour la distance entre Pont de Claix et la ville
<code>boolean</code>	<code>setVille(java.lang.String ville)</code> cette méthode met à jour la ville associée au trajet

Methods inherited from class java.lang.Object

`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

Constructor Detail

Trajet

```
public Trajet()
```

Method Detail

getDistance

```
public int getDistance()
```

cette méthode permet de récupérer la distance du trajet en kilomètres

Returns:

la fonction retourne la longueur du trajet en kilomètres

setDistance

```
public boolean setDistance(int distance)
```

cette méthode met à jour la distance entre Pont de Claix et la ville

Parameters:

`distance` - est la distance entre le ville et Pont de Claix (mairie à mairie)

Returns:

la fonction retourne vrai si la mise à jour a été possible

getVille

```
public java.lang.String getVille()
```

cette méthode donne la ville concernée par le trajet

Returns:

la fonction retourne le nom de la ville

setVille

```
public boolean setVille(java.lang.String ville)
```

cette méthode met à jour la ville associée au trajet

Parameters:

ville - est le nom de la ville d'arrivée

Returns:

la fonction retourne vrai si le nom de la ville a été mis à jour

Voici le code java à partir duquel ce document a été produit :

```
/**
 * Cette classe décrit un trajet à partir de Pont de Claix.
 *
 * @version 1 Created on 4 janv. 2005
 *
 * @author corre
 *
 */
public class Trajet
{
    private int m_nDistance; // distance en kilomètres du trajet
    private String m_strVille; // ville d'arrivée

    /**
     *
     * cette méthode permet de récupérer la distance du trajet en kilomètres
     * @return la fonction retourne la longueur du trajet en kilomètres
     */
    public int getDistance()
    {
        return m_nDistance;
    }

    /**
     *
     * cette méthode met à jour la distance entre Pont de Claix et la ville
     * @param distance est la distance entre le ville et Pont de Claix (mairie à mairie)
     */
}
```

```

    * @return la fonction retourne vrai si la mise à jour a été possible
    */
    public boolean setDistance(int distance)
    {
        if (distance > 0)
        {
            m_nDistance = distance;
        }
        return distance > 0;
    }
    /**
     *
     * cette méthode donne la ville concernée par le trajet
     * @return la fonction retourne le nom de la ville
     */
    public String getVille()
    {
        return m_strVille;
    }
    /**
     *
     * cette méthode met à jour la ville associée au trajet
     * @param ville est le nom de la ville d'arrivée
     * @return la fonction retourne vrai si le nom de la ville a été mis à jour
     */
    public boolean setVille(String ville)
    {
        if ((ville == null) || (ville.trim().equals(""))) // nom de ville incorrect
        {
            return false;
        }
        else
        {
            m_strVille = ville;
            return true;
        }
    }
}

```

Et enfin voici un petit programme d'essai très simple :

```

public class Main
{
    public static void main(String[] args)
    {
        Trajet t = new Trajet();
        t.setDistance(10);           // ici je ne teste pas les booléens en retour,
        t.setVille("Grenoble");      // c'est mon ( mauvais ) choix de programmeur,
                                    // et j'en assumerai les conséquences.
    }
}

```

Exercice 3 : où l'on découvre les bienfaits de l'encapsulation

Attention le chapitre 4 se fera à partir du résultat du chapitre 2. Pensez donc bien à copier les fichiers d'un répertoire à un autre, puis de construire le nouveau projet sur ce nouveau répertoire. Ainsi vous retrouverez le projet du chapitre 2 comme vous l'avez laissé.

- Dans cette nouvelle version du programme, l'adresse devra être décortiquée pour que l'on puisse exploiter séparément le numéro de rue, le nom de la rue, le code postal et la localité. Ces informations seront rangées dans des données membres séparées qui remplaceront la donnée membre `m_strAdresse`.
- Cette modification devra être transparente pour l'utilisateur : les interfaces et les fonctionnalités de l'**objet Individu** ne devront pas être modifiées, seulement enrichies (ce que l'on appelle une compatibilité ascendante).
- L'interface de programmation de la fonction **lire** ne sera pas modifiée. Mais la saisie du numéro de rue, du nom de rue, du code postal et de la ville se fera indépendamment : si un champ est erroné, il sera ressaisi avant de passer au champ suivant. (ici nous apportons une modification du comportement de la méthode `lire`, afin de la rendre plus agréable pour l'usage de l'opérateur, mais d'un strict point de vue technique, ce n'était pas nécessaire)
- Le code de la fonction **setAdresse** sera revu pour faire appel à des fonctions élémentaires correspondant aux nouveaux champs (**setNum**, **setRue**, **setCodePostal**, **setVille**), qui effectueront les vérifications utiles : le code postal est numérique et comprend 5 chiffres ; le numéro de rue est numérique ; les noms de rue et de ville ne comprennent que des lettres et des espaces, et est non vide. L'adresse sera rentrée sous le format suivant : « 123, impasse René FONCE, 39820, Grolles la Jolie ». La virgule servant de séparateur. Vous utiliserez la méthode **split** de la classe `String` pour découper la chaîne en morceaux qui seront traités par la méthode `set` élémentaire adéquate, et vous utiliserez la méthode **trim** de la classe `String` pour enlever les espaces excédentaires. La valeur de retour booléenne de la méthode **setAdresse** devrait nous être utile ici.
- Le code de la méthode **getAdresse** sera lui aussi réécrit, en utilisant les méthodes `get` élémentaires.
- La donnée membre privée `m_strAdresse` disparaît au profit de 4 données membres privées : `m_nNumRue`, `m_strNomrue`, `m_nCodePostal`, `m_strVille`.

Conseils

- Pensez à stocker les nouvelles données membres dans les types JAVA les plus judicieux, et rappelez-vous que les méthodes `get` et `set` travaillent en respectant les types des attributs.
- Dans cette version, on ne sait pas encore gérer les exceptions JAVA : il faut donc vérifier que les conversions sont possibles avant de les effectuer (exemple : conversion du code postal et du numéro de rue en `int`).
- Faire deux fonctions privées : **boolean estEntier (String s)** ; et **int convertir (String s)** ; qui seront utilisées dans la conversion des codes postaux et des numéros de rues en entier.

- N'oubliez jamais de vous inspirer du code existant et d'appeler toutes les fonctions membres existantes qui pourraient simplifier le codage (exemple la fonction estAlpha que l'on a réalisée précédemment) ! Faire un modèle unique de fonction que vous suivrez pour coder toutes les fonctions élémentaires setRue, setNum, setCodePostal... Cela peut paraître fastidieux mais je l'ai fait volontairement pour vous inciter à factoriser le plus possible votre code, en utilisant des petites fonctions privées à la classe, ou des méthodes de boîte à outils.
- En final, vérifiez que la modification de la classe n'entraîne pas de modifications du programme principal
- N'oubliez pas la mise à jour de la documentation technique.
- Mettre en place les tests automatiques des classes développées (ou tests de non régression). Pour cela consultez la documentation en ligne sur JUNIT par exemple www.vogella.de/articles/junit/article.html .

Exercice 4 : comment créer un objet cohérent dès sa naissance.

- **Pour simplifier le codage, on repartira de la première version écrite en objet de la classe Individu (sans le traitement des adresses)**
- Faisons la suite d'instructions suivantes:

```
Individu lePire = new Individu();  
System.out.println(lePire.getNom());
```

Que se passe t'il à l'exécution de ce bout de programme? Pourquoi?

Nous avons sécurisé notre objet depuis le moment de sa lecture jusqu'à sa mort. Mais entre le moment de sa naissance (new), et sa première initialisation cohérente, il n'est pas sécurisé. Notre problème est donc simple, notre objet doit naître sécurisé. Tout se joue maintenant au moment du new, à la naissance de l'objet. Ainsi notre objet sera sécurisé de sa naissance jusqu'à sa mort.

- Comment sécuriser notre objet? (voir Limouzin Java2 chapitre 3 : en quoi son constructeur d'initialisation n'est il pas sécurisé ?)
- Faire un constructeur d'initialisation pour la classe individu.
- Le constructeur par défaut est-il toujours disponible? Pourquoi?
- Faire un constructeur par défaut (vous pouvez utiliser le this(paramètres) qui permet d'appeler en première instruction d'un constructeur, un autre constructeur de la classe).

Conseils :

- Pour chaque classe que l'on définira il faudra bien réfléchir aux différents constructeurs que l'on créera, si nécessaire. Il n'y a pas de règle générale pour connaître le nombre de constructeurs à créer, c'est le problème qui vous l'imposera. L'analyse du problème est donc importante pour pouvoir bien définir ses objets. C'est ce que vous verrez en UML.
- La classe String possède 11 constructeurs différents. D'autres classes n'en possèdent pas (c'est-à-dire qu'elles n'ont que celui défini par défaut) par exemple la classe Applet, ou la classe Trajet montré en exemple dans l'exercice 2 (vous noterez que le constructeur par défaut absent du code, est pourtant documenté par javadoc). D'autres classes ont des constructeurs rendus inaccessibles (constructeurs privés) par exemple les classes qui ne sont que des bibliothèques de fonctions comme la classe Math (vous ne pouvez pas construire un objet de la classe Math).

Exercice 5 : comment stocker et gérer une liste d'objets d'une même classe...

Nous allons maintenant gérer un tableau d'objets. Rien de neuf à l'horizon, puisque nous avons déjà fait un tableau de stagiaires dans le chapitre 9 de java procédural. Et pourtant ...

- Dans cette nouvelle version du programme, on veut gérer plusieurs individus qui peuvent être par exemple des salariés d'une entreprise, un groupe de stagiaires de formation, ... Nous traiterons un groupe d'individus.
- Le programme devra saisir au clavier les caractéristiques d'au plus MAX individus, en arrêtant la saisie à la demande de l'opérateur, puis il réaffichera l'ensemble des individus saisis.

Conseils :

- Pour stocker le groupe d'individus, on utilisera un tableau (au sens Java) de MAX individus au plus.
- Définir MAX (notez qu'une constante s'écrit en majuscule) comme une constante de type **int**, avec les mots clé **final** et **static**.
- Attention aux spécificités des tableaux en Java : ce sont des objets qui ne peuvent pas être dimensionnés à la déclaration, mais qui doivent l'être à la création

Exemple : **Individu tab [] ;**
 tab = new Individu [MAX] ;

- La création du tableau ne crée pas les objets contenus mais seulement des références : il faut ensuite penser à créer chaque individu, dans la boucle de saisie du groupe d'individus:
tab [i] = new Individu () ;
- Pour permettre à l'opérateur de décider de la fin de la saisie, rajouter une fonction booléenne **veutContinuer** aux utilitaires de la classe Saisie
- Ajouter une fonction permettant d'afficher l'ensemble des individus du tableau.
- Quand notre programme fonctionne, nous constatons qu'un groupe d'individus a des caractéristiques (attributs), un savoir faire (méthodes), en un mot a toutes les caractéristiques d'un objet. Nous allons donc en faire une classe Groupe, avec ses méthodes publiques, ses attributs privés, ses constructeurs, et ses méthodes get et set nécessaires. Nous allons avoir un attribut **m_nCpt** qui compte le nombre d'individus du groupe. Est-il prudent de faire une méthode **setCpt** publique ? Est-il même commode d'utiliser cette méthode pour incrémenter (dans la classe) le nombre d'individus ? Ceci nous montre qu'il faut bien réfléchir à l'interface de notre objet (donc à réaliser la documentation) avant de coder l'objet. La documentation vous donne le point de vue d'un programmeur qui utilise votre objet, et il est important pour le développeur d'objets de prendre aussi ce point de vue.
- Vous complétez la classe avec les méthodes lire et afficher, puis avec getMax (nombre maximum d'individus dans le groupe), getNb (nombre d'individus effectivement dans le groupe), individuAt (retourne une référence sur l'individu dans la case du tableau visée (voir charAt de la classe String) ou null si la case est non occupée.

- Vous aurez 2 constructeurs pour la classe Groupe. Celui par défaut crée un Groupe de 20 individus, celui par initialisation crée un groupe du nombre d'individus donné à la construction.
- Vous documenterez, cela va sans dire, votre classe Groupe.

Ainsi nous allons commencer à réfléchir objet : nous voilà de plein pied dans le monde des objets.

Exercice 6 : comment stocker et gérer une liste d'objets d'une même famille...

- Dans cette nouvelle version du programme, nous voulons gérer plusieurs types de personnes dans l'entreprise.
- Nous avons défini un individu de manière générale. Vous commencerez par définir une méthode **toString** pour la classe Individu (voir les conseils). Dans l'entreprise nous avons des salariés qui sont des individus qui ont un salaire (type float) et un statut (type int). Nous avons également des commerciaux qui sont des salariés mais qui ont en plus un chiffre d'affaire (type float), et un ratio (type float) sur ce chiffre d'affaire. cela permet de calculer leur salaire en y incluant les intéressements aux affaires qu'ils apportent. Et enfin nous avons des clients qui sont des individus qui doivent des sous à l'entreprise (type float).
- Nous allons donc créer la classe des salariés, avec les fonctions polymorphes associées (voir poly Limouzin Java2 chapitre 4).
- Nous allons ensuite créer la classe des commerciaux, et la classe des clients.
- Nous allons enfin construire un tableau contenant des salariés, des commerciaux et des clients. Nous allons afficher l'ensemble des individus contenus dans le tableau. Nous allons pour cela reprendre la classe Groupe, dans laquelle la ligne **m_tab[i] = new Individu()** ; de la méthode **lire()** sera remplacée par une instruction **switch** pour créer à la demande de l'utilisateur soit un individu, soit un salarié, soit un commercial, soit un client. Cela nécessite t'il de modifier la fonction d'affichage d'un tableau d'individus définie dans l'exercice 5 ?

Conseils :

- La notion que l'on aborde ici est la notion d'**héritage**. C'est la deuxième grande caractéristique des langages de programmation objet, après l'encapsulation. L'héritage permet de classer les objets, de factoriser leur comportement, et de définir des comportements "mutants". Le travail d'un développeur, en programmation objet se trouve là: héritage d'une classe ayant un comportement proche, puis définition des comportements ayant muté.
- Vous ne pouvez pas faire une impasse sur cette notion, sans faire une impasse sur la programmation objet. (nous allons y revenir grandement).
- Pour y voir plus clair dans notre démarche, nous allons développer pour chaque classe une interface (description des intentions de la classe, ou contrat d'opération), puis les classes abstraites nécessaires à la construction de notre arborescence. N'oubliez pas le principe, les classes terminales d'une arborescence de classe sont concrètes, les classes non terminales sont abstraites.
- Transformez la classe Individu en classe abstraite (abstract) et définissez la classe Personne comme une classe concrète dérivant d'Individu. Définissez également l'interface de la classe Individu et de la classe personne.
- Développez la classe Salarié complètement, avec sa documentation, et faites-la valider avant de développer les autres classes.
- Toute classe a une méthode **toString()** qui permet à une de ses instances de donner son équivalent en chaîne de caractère. C'est un comportement polymorphe, c'est à dire qu'il évoluera à chaque fois que la classe évoluera. Développez cette méthode dans la classe Individu et dans toutes les classes dérivées. Utilisez-la dans la méthode afficher de la classe Individu. Cette méthode sera appelée partout où java à besoin d'une chaîne de caractères, et ne trouve qu'un objet. Exemple :

```
Personne gus = new Personne () ;  
System.out.println(gus) ; // identique à System.out.println(gus.toString()) ;
```

- N'oubliez pas de typer les méthodes get et set avec le type de l'attribut.

- Le message **getSalaire()** est la question : « combien gagnes-tu ? ». A cette question un salarié et un commercial ne répondront pas de la même façon. Un salarié donne son salaire fixe, un commercial donne son fixe additionné avec sa commission. Le comportement de la méthode **getSalaire** a donc muté pour le commercial. Il faudra donc faire un polymorphisme de la méthode **getSalaire** pour le commercial.

Exercice 7 : Synthèse sur des exercices faits en java procédural

Nous allons reprendre deux exercices que nous avons réalisés en java procédural : la gestion d'une pile d'entiers, et la gestion d'une table de noms classés alphabétiquement. Le but est ici de faire la synthèse sur l'encapsulation en créant les classes Pile et TableAlpha. Les procédures d'initialisation seront effectuées dans les constructeurs des classes, le code est à peu de chose près le même, simplement il faut intégrer les fonctions dans les classes pour en faire des méthodes de classe. Au préalable vous aurez déterminé les attributs de ces classes, c'est-à-dire les informations qui représentent successivement une pile et une table classée alphabétiquement, puis vous aurez construit l'interface de ces classes, c'est-à-dire que vous aurez construit la documentation de ces classes.

Vous referez également les programmes de test de ces classes.

Ce travail est un travail de modification du code existant, qui ne demande pas énormément de frappe au clavier. Mais ceci vous permettra de voir si vous avez bien cerné la notion d'objet dans sa composante encapsulation. Faites bien attention à ne pas confondre le comportement de l'objet, et un usage particulier de l'objet.

Les programmes java procéduraux sont fournis dans les répertoires Pile et TableAlpha.

Exercice 8 : Synthèse sur la notion d'héritage

Avant de développer nos connaissances sur le langage java, prenons le temps de développer une application mettant en œuvre les notions d'héritage que nous utiliserons à outrance désormais.

Nous allons développer des classes qui permettront ultérieurement de réaliser un logiciel de supervision. Qu'est ce que la supervision ? Prenons l'exemple de surveillance de locaux dont les accès sont particulièrement contrôlés, et dont la sécurité doit être maximale. Un poste de surveillance regroupe sur un ou plusieurs écrans toutes les informations sur les entrées sorties de personnes, sur les différentes alarmes pouvant se produire (détecteurs de fumées, alarme anti-intrusion, ...), sur l'éclairage des locaux, etc. ... Dans la salle de contrôle, 24h / 24, un agent de sécurité est chargé, suivant les informations portées à sa connaissance, d'agir, ou de faire agir les personnes compétentes en cas de problème de sécurité. Le logiciel gérant les informations de sécurité est appelé un logiciel de supervision. (les logiciels de supervision concernent particulièrement les domaines sensibles, nucléaire, chimie, laboratoires classés, mais aussi les chaînes de fabrication, la distribution d'électricité, le suivi des colis en express, ...).

Revenons à notre supervision de locaux. Quand une personne pénètre dans une salle, à l'aide de son badge, l'opérateur en est averti sur son écran de contrôle. Un message d'information est envoyé automatiquement au poste du contrôleur. Les informations du message permettent de savoir que M Dupont est entré dans la salle n°435 à 16h45 le 22 janvier 2005.

Tous les accès ont des alarmes (portes et fenêtres) et des détecteurs de fumée et d'eau équipent chaque salle.

Quand un défaut survient, une alarme arrive automatiquement sur le poste de l'opérateur. L'opérateur doit alors prendre en compte cette alarme (il peut être occupé à traiter un autre défaut par ailleurs, et ne peut pas forcément prendre en compte l'alarme immédiatement). Puis, quand le problème est traité, il doit solder l'alarme, c'est-à-dire la désactiver.

Les messages servent à tracer l'historique des événements qui ont précédés une alarme, et quelques fois à comprendre la cause de celle-ci.

1) Dans le cadre de développement d'un logiciel de supervision, nous allons développer une classe d'objets message définie comme suit :

- Une chaîne représentant le type de message.
- Une chaîne représentant le libellé du message
- Une chaîne représentant la provenance du message
- Une date et heure représentant le moment de création du message
- Une fonction permettant d'afficher le message sur l'écran (surtout sans utiliser println).
- Les fonctions d'accès aux données membres.
- Et le constructeur de message par initialisation.

Les dates et heures seront représentées sous forme Mardi 05 Octobre 1999 13h38mn21s340. En java, la datation a été traitée dans un premier temps comme référence à un calendrier universel, et tant qu'à faire, au calendrier occidental (calendrier Grégorien). Mais l'ouverture au monde, de java, a conduit à construire d'autres classes que la classe Date pour prendre en compte la multiplicité des calendriers.

La classe abstraite Calendar décrit le comportement d'une datation interprétée au travers d'un calendrier. La classe GregorianCalendar est une classe héritée de Calendar qui interprète les dates par rapport au calendrier grégorien. D'autres calendriers seront développés pour interpréter les différents calendriers lunaires ou lunisolaires que l'on trouve dans le monde.

Quand on crée une instance de la classe GregorianCalendar, nous disposons de la datation système du moment de la création de l'objet interprété par le calendrier grégorien, mais disponible par défaut en Anglais. Nous verrons plus tard la gestion des défauts. Nous allons donc créer une classe qui est un GregorianCalendar mais qui propose la date et heure en français, en concaténant les différentes informations composant une date et une heure, grâce à la fonction get de la classe Calendar.

get (Hour) vous retourne l'heure

get (MONTH) vous retourne une constante qui vaut soit JANUARY, soit FEBRUARY, ... qui sont des constantes de la classe Calendar, dont je ne veux pas connaître la valeur car :

- ça ne m'intéresse pas.
- ces valeurs pourraient changer, et je ne veux pas en être dépendant.

(pour plus de précisions voir la documentation des classes concernées)

Vous réaliserez cette nouvelle classe DateFR, que vous documenterez et que vous testerez (avant et après midi , ...).

Vous réaliserez, et testerez votre classe Message, que vous aurez au préalable documentée.

2) Pour pouvoir gérer l'ensemble des messages, nous utilisons une collection (un vecteur ici). Le vecteur est une collection permettant de conserver des informations d'un type donné. Par exemple :

```
Vector<Joueur> équipe = new Vector<Joueur>();
```

Equipe est un vecteur contenant uniquement des objets de type Joueur.

Vector est une classe générique, car quand nous l'utilisons, nous spécifions le type des objets qui seront contenus dans le Vecteur.

Voici la documentation du vecteur (extrait et traduit):

```
public Vector(int initialCapacity,  
               int capacityIncrement)
```

Construit un Vector vide avec la taille initiale initialCapacity , et une capacité d'agrandissement de capacityIncrement .

Paramètres:

initialCapacity – la capacité initiale du Vector.

capacityIncrement – la taille d'accroissement du Vector, quand un élément est ajouté alors que le Vector est plein.

Throws:

IllegalArgumentException – si la capacité initiale est négative.

```
public void addElement(E élément)
```

Ajoute un élément à la fin du Vector, incrémentant sa taille de un. La capacité du Vector est agrandie si la taille du Vector devient supérieure à sa capacité. E est la classe des éléments contenus dans le vecteur.

Paramètres:

obj – l'objet à ranger dans le Vector.

```
public E elementAt(int index)
```

Retourne l'élément à une position donnée du Vector.

Paramètres:

`index` - index de l'élément à retourner.

Throws:

`ArrayIndexOutOfBoundsException` - index hors bornes (`index < 0 || index >= size()`).

`public int size()`

Retourne le nombre d'éléments effectivement contenus dans le `Vector`..

Returns:

Le nombre d'éléments contenu dans le `Vector`.

Donnez le code permettant un affichage simple de l'ensemble des messages contenus dans une liste. Pensez à utiliser une itération appropriée pour parcourir votre liste.

3) Les alarmes sont des messages un peu particuliers : ils ont

- Un niveau de gravité de 1 à 5. Si le niveau est inférieur à 1, il sera mis à 1, et s'il est supérieur à 5, il sera mis à 5.
- Une date et heure de prise en compte de l'alarme par un opérateur.
- Nom de l'opérateur ayant pris en compte.
- Une date et heure de solde de l'alarme par un opérateur.
- Un nom d'opérateur ayant soldé l'alarme.

Définissez la classe alarme qui permet de :

- Construire un message d'alarme.
- Prendre en compte un message d'alarme.
- Solder un message d'alarme.
- Afficher un message d'alarme sur la sortie standard. Les informations de prise en compte et de solde ne seront affichées que si nécessaire.

Vous documenterez et testerez votre classe alarme. Vous testerez particulièrement que l'on ne peut solder qu'une alarme qui a déjà été prise en compte, que l'on ne peut pas prendre en compte deux fois une alarme, ...

Que cela change t'il si nous désirons afficher une liste (dans un vecteur) d'alarmes et de messages par rapport à la question 2° ?

Vous testerez votre réponse.

Note : nous ne savons pas comment le logiciel de supervision récupère le nom de l'opérateur pour solder ou prendre en compte une alarme (donne t'il son nom à chaque fois ? badge t'il ? se connecte t'il lors de sa prise de fonction, et ce nom de connexion est il utilisé ?). Cela dépend de la conception du programme de supervision. Nos deux méthodes auront donc en paramètre le nom de l'opérateur, à la charge de la supervision de récupérer ce nom, comme elle le doit. Nos méthodes vérifieront toutefois que le nom fournit est non **null**, non vide, et alphabétique.

4) Nous avons créé les messages sans filtrer les informations à l'origine de la création du message. Nous allons maintenant poser les conditions suivantes : le libellé, la provenance et le type du message doivent être des chaînes de caractères non **null**, non vides et ne contenant pas que des espaces ou retour chariot. Si tel n'est pas le cas, nous ne voulons pas mettre des valeurs stupides dans le message, mais nous voulons refuser la création du message. Pour cela nous allons gérer le mécanisme d'exception.

Ce mécanisme d'exception se gère en trois temps :

- Création d'une classe dérivant de la classe Exception.
- Quand les conditions sont remplies, création d'une instance de cette classe d'exception, et lancement de cette instance (une exception est donc un objet que l'on lance).
- Récupération de l'exception par le programme qui est à l'origine de cette gestion d'exception, et traitement adéquat, ou fin d'application suivant les cas.

Vous étudierez les exceptions dans le polycopié de Limouzin, en tenant compte de quelques remarques :

- La classe Exception contient déjà une information textuelle servant de message d'erreur.
- Vous pouvez ajouter à votre nouvelle exception toutes les informations que vous désirez, grâce au mécanisme d'héritage. Mettez par contre en place les méthodes get associées à chacune de ces informations.
- Chaque fois que l'on pourra, il sera préférable de garder les méthodes set avec un retour booléen, plutôt que de lui faire lancer des exceptions.
- Méfiez-vous du fait de donner en donnée membre de l'exception l'objet qui est lui-même à l'origine de l'exception. Si l'exception se produit à la construction de l'objet, l'objet ainsi transmis à l'exception n'est pas stable, et cela peut nous amener des ennuis.

Vous allez créer la classe MessageException dérivant de la classe Exception. La création d'un message avec des informations incorrectes générera une exception. Le programme de test récupérera cette exception pour afficher un message ou saisir à nouveau les informations de création du message.

Cela entraînera aussi des modifications de comportement des alarmes qui sont des messages, donc auront également à gérer les exceptions.

Vous générerez également une exception quand l'utilisateur essaye de récupérer un message inexistant dans votre collection (messageAt). L'exception ArrayIndexOutOfBoundsException semble la bonne exception à lancer. La méthode messageAt ne nécessite pas de clause throws : pourquoi ?

Faites bien la distinction entre les exceptions utilisateur (qui ont un sens dans le métier de l'utilisateur) et les exceptions de mise au point.

Les exceptions utilisateur nécessitent un traitement, ont obligatoirement une clause throws, et dérivent souvent de la classe exception. Elles ont un sens dans la logique du métier du client.

Les exceptions de mise au point ne nécessitent pas forcément un traitement (ce qui entraînera un arrêt du programme pour mise au point), n'ont pas forcément une clause throws, et héritent généralement de la classe RuntimeException. Elles n'ont pas de sens dans la logique du métier du client.

Deuxième partie : réalisation d'interfaces avec AWT

Découverte des applets.

1) Créez un nouveau projet sous Eclipse. Construisez une classe dérivant de la classe `java.applet.Applet`. Créez les fonctions `init`, `start`, `stop`, `paint`, `destroy` mais faites des fonctions vides. Dans un fichier ayant l'extension `html`, vous insérez les lignes suivantes :

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="ma petite applette">
</HEAD>
<BODY>

<P>&nbsp;  </P>

<applet code="applette.class" name=applette id=applette width=320
height=200 VIEWASTEXT>
</applet>
</BODY>
</HTML>
```

Le nom `applette` sera remplacé par le nom de votre classe.

L'applet que vous créez est caractérisée par cinq fonctions qui sont `init()`, `start()`, `stop()`, `destroy()`, et `paint()`. Ces fonctions sont les fonctions qui définissent le comportement standard d'une applet. Cherchez dans la documentation le rôle de chacune de ces fonctions. Nous allons le vérifier par programme :

Créez un compteur par fonction initialisé à zéro. (`cptinit`, `cptstop`, `cptstart`, ...)

Dans chaque fonction incrémentez le compteur concerné.

Dans la fonction `paint` affichez la valeur des cinq compteurs (pour cela regardez la fonction de la classe `Graphics` utilisable).

Générez le projet.

Lancez la page `html` (ici il faut repasser par votre explorateur favori) et faites la vivre (mise en icône, lien avec une autre page et retour, ...) en observant la progression des compteurs. Les explorateurs ne respectent pas toujours le principe de fonctionnement des applets défini en des temps déjà reculés. C'est bien le cas de `IE x` (avec $x > 3$).

D'autres navigateurs respectent scrupuleusement le rôle de chacune de ces méthodes.

2) Nous allons créer une applet paramétrable. Soit une applet qui affiche un texte. Nous voulons pouvoir passer en paramètre : le texte à afficher, la couleur d'écriture, la taille de la police, la police de caractères, le style.

Pour cela voir les classes `Graphics` et `Font`, et la documentation . (`Visual J++` page 249 et `Java 1.1` chap. 9 (police et couleur))

Vous utiliserez des paramètres dans la page `HTML` suivant le modèle suivant :

```
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="ma petite applette">
</HEAD>
<BODY>
```

```
<P>&nbsp;</P>
```

```
<applet code="applette.class" name=applette id=applette width=320
height=200 VIEWASTEXT>
<param name="txt" value="Ceci est un texte">
<param name="color" value="FADCE4">
<param name="size" value="20">
<param name="font" value="arial">
<param name="style" value="bold">
</applet>
</BODY>
</HTML>
```

Vous sécuriserez la récupération des paramètres : le programmeur qui insère votre applet dans sa page html peut oublier un paramètre (ou se tromper dans son nom, ce qui revient au même). Il peut aussi se tromper dans la valeur fournie pour un paramètre. Vous prévoirez donc des valeurs par défaut pour traiter ces différents cas.

Par exemple la valeur associée à la couleur va être traitée de la manière suivante : la chaîne représente en code hexa la valeur de chaque composante RVB de la couleur désirée (RRVVBB). Vous allez avoir un code qui va ressembler à ceci :

```
Color c ;
try
{
    String s = getParameter("color");    // récupération du paramètre
    int val = Integer.parseInt(s,16);    // conversion d'une chaîne hexadécimale en entier
    c = new Color( val ) ;                // construction de la couleur
}
catch ( Exception e)                    // s est null ou s n'est pas une chaîne hexa, ou
                                        // val ne définit pas une couleur
{
    c = Color.BLACK ;                    // une couleur par défaut
}
```

Vous avez vu le principe qui régit les applets. Tous les exercices qui vont suivre seront des applications, mais il serait simple d'en faire des applets.

Une petite aide pour connaître les polices utilisables dans une application java :

Comment récupérer les noms des fontes reconnues par un système :

Il est des fois nécessaire de connaître les noms de polices qui vont pouvoir être interprétées par le système, et donc par Java. Par exemple dans le cas d'une applet devant modifier la police qu'elle utilise en fonction des paramètres que lui passe la page Web la lançant.

Pour lister les polices acceptées :

```
// Création d'un environnement graphique
// Et récupération de son environnement
GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();

// Fonction retournant un tableau de String constitué des noms de
polices
String [] tab = ge.getAvailableFontFamilyNames();
```

```
// Boucle d'affichage
for ( String s:tab)
{
    System.out.println(s);
}
```

Traitement des événements

1) Principes généraux :

En Java 1.1 et 2 nous allons créer pour chaque événement qui peut se produire (clic d'un bouton, déplacement de la souris, ...) un objet écouteur, qui sera chargé d'effectuer les traitements adéquats quand l'événement écouté se déclenche. Il y aura donc autant d'écouteur que d'événements écoutés. Chaque écouteur sera un objet lui-même, ce peut être un objet spécial dont le rôle ne sera qu'écouter, mais aussi un objet conteneur, ou l'application. Nous examinerons chacun des cas sur un exemple très simple dans le **chapitre 2**.

Quand on crée un objet spécial pour écouter un événement trois cas peuvent se produire :

- Nous créons un objet comme nous savons le faire.
- Nous créons un objet interne à une classe existante (par exemple la frame qui contient le contrôle déclenchant l'événement).
- Nous créons un objet anonyme interne à une classe existante (par exemple la frame qui contient le contrôle déclenchant l'événement).

Chacun de ces trois cas sera également examiné.

Nous allons regarder comment cela fonctionne. Chaque component a une fonction **processEvent**:

- La fonction processEvent ne retourne rien, car elle ne traite pas la remontée (ou la propagation) des événements. Le ou les composants qui désirent écouter un événement créent chacun un écouteur et s'enregistrent auprès du composant à écouter
- La fonction processEvent traite toutes les familles d'événements, il ne sera donc jamais nécessaire de redéfinir cette fonction. Ce qui n'empêche pas les programmeurs de le faire pour des raisons qui peuvent sembler obscures (écrire un peu moins de ligne, ...).

En bref pour écouter un événement, il suffit, à la classe écouteur, de s'abonner au service d'écoute de ce type d'événement, auprès du composant concerné, puis il faut développer la fonction (ou les fonctions) qui sera activée quand le service est demandé.

Dans le **chapitre 3** il est défini, pour chaque événement, une liste de fonction de traitement. Il est défini de même, pour chaque composant de l'interface, les événements que l'on peut recevoir.

Au **chapitre 4** nous verrons comment simplifier l'écriture d'écouteur dans certains cas, grâce aux adaptateurs.

Le **chapitre 5** est une progression d'exercices mettant tout cela en musique.

2) Exemples d'écouteurs d'événements

Nous allons voir la manière de traiter un événement simple. Nous allons mettre un bouton dans une fenêtre, et quand nous cliquons le bouton, la fenêtre lui fait changer de nom. Nous allons donc mettre en place un service d'écoute sur un bouton.

Nous allons décliner cet exemple suivant les différentes manières de le résoudre. Tous les exemples qui suivent ont été tirés de la littérature, ou du net, et représentent la jungle dans laquelle vous aller devoir vous mouvoir quand vous allez traiter des exemples. Dans tous les cas vous devrez vous ramener à un des deux premiers cas qui sont les seules manières de produire un travail propre, accessible à tout le monde et réutilisable.

- 1) Nous allons travailler avec une classe de test, une classe de frame et une classe écouteur externe (théoriquement la solution la plus belle).
- 2) Nous allons travailler avec une classe de test, une classe de frame et une classe écouteur interne (pratiquement la solution la plus utilisée).
- 3) Nous allons travailler avec une classe de test, une classe de frame et une classe écouteur interne anonyme (perte de lisibilité du programme).
- 4) Nous allons travailler avec une classe de test, une classe de frame et une classe écouteur interne anonyme sur laquelle sera définie une instance anonyme (grosse perte de lisibilité).
- 5) Nous allons enfin travailler avec une classe de test, et une classe héritée de frame qui traite le clic du bouton (ici c'est la frame qui est aussi l'écouteur. Nous nous éloignons des principes objet).
- 6) Nous allons travailler à la manière sale en oubliant les principes objets : une seule classe fait tout : le main, la fenêtre et l'écouteur (cet exemple se trouve, hélas, souvent dans la littérature).

Notons que tous ces exemples résolvent le même problème. Voyons-les, puis nous ferons le point.

// 1) création d'une classe externe qui écoute l'événement action du bouton

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class Fen extends Frame
{
    private Button b = new Button("oui");
    public Fen()
    {
        add(b);
        Ecouteur oreille = new Ecouteur(b);
        // on crée un écouteur pour gérer les actions sur le bouton
        b.addActionListener(oreille);
        // l'écouteur s'enregistre ( oreille ) comme un écouteur des
        // actions sur le bouton b.
        setBounds(200,200,100,100);
        setVisible(true);
    }
}

public class Main           // ici classe de test de la classe Fen
                           // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )
{
    public static void main( String [] args)
    {
        Fen mafen = new Fen();
    }
}

class Ecouteur implements ActionListener
                           // implémentation de l'interface ActionListener car mon écouteur
                           // est un écouteur d'événements sur les actions sur les contrôles.
{
    private Button b;
    public Ecouteur(Button bouton)
    {
        b = bouton;    // on récupère la référence sur le bouton car étant une classe
                        // externe on ne connaît pas le bouton
    }
    public void actionPerformed ( ActionEvent evt )
        // mon écouteur étant un écouteur d'ActionEvent, il lui faut
        // une fonction actionPerformed qui sera activée quand le
        // bouton sera cliqué.
    {
        b.setLabel("non");
    }
}
```

// 2) création d'une classe interne qui écoute l'événement action du bouton

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class Fen extends Frame
{
    private Button b = new Button("oui");
    public Fen()
    {
        add(b);
        Ecouteur oreille = new Ecouteur();
        // on crée un écouteur pour gérer les actions sur le bouton
        b.addActionListener(oreille);
        // l'écouteur s'enregistre ( oreille ) comme un écouteur des
        // actions sur le bouton b.
        setBounds(200,200,100,100);
        setVisible(true);
    }
    class Ecouteur implements ActionListener
        // implémentation de l'interface ActionListener car mon écouteur
        // est un écouteur d'événements sur les actions sur les contrôles.
        // nous connaissons le bouton b car nous sommes dans la classe
        // nous avons ainsi la connaissance de tous les composants de la
        // classe
    {
        public void actionPerformed ( ActionEvent evt )
            // mon écouteur étant un écouteur d'ActionEvent, il lui faut
            // une fonction actionPerformed qui sera activée quand le
            // bouton sera cliqué.
        {
            b.setLabel("non");
        }
    }
}

public class Main          // ici classe de test de la classe Fen
                           // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )
{
    public static void main( String [] args)
    {
        Fen mafen = new Fen();
    }
}
```


// 3) création d'une classe interne anonyme à la frame qui écoute l'événement action du bouton

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class Fen extends Frame
{
    private Button b = new Button("oui");
    public Fen()
    {
        add(b);
        ActionListener ecouteur = new ActionListener()
            // implémentation de l'interface ActionListener
            // car mon écouteur est un écouteur d'événements
            // sur les actions sur les contrôles.
            // ici on crée une classe interne à la classe et anonyme
        {
            public void actionPerformed ( ActionEvent evt )
                // mon écouteur étant un écouteur d'ActionEvent, il lui faut
                // une fonction actionPerformed qui sera activée quand le
                // bouton sera cliqué.
            {
                b.setLabel("non");
            }
        };
        b.addActionListener(ecouteur);
            // l'écouteur s'enregistre ( ecouteur ) comme un écouteur des
            // actions sur le bouton b.
        setBounds(200,200,100,100);
        setVisible(true);
    }
}

public class Main           // ici classe de test de la classe Fen
                            // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )
{
    public static void main( String [] args)
    {
        Fen mafen = new Fen();
    }
}
```

// 4) création d'une classe interne anonyme à la frame qui écoute l'événement action du bouton (écriture condensée)

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class Fen extends Frame
{
    private Button b = new Button("oui");
    public Fen()
    {
        add(b);
        b.addActionListener( new ActionListener()
            // implémentation de l'interface ActionListener
            // car mon écouteur est un écouteur d'événements
            // sur les actions sur les contrôles.
            // ici on crée une classe interne anonyme à la classe,
            // et, dans la foulée, une instance anonyme sur cette classe
        {
            public void actionPerformed ( ActionEvent evt )
                // mon écouteur étant un écouteur d'ActionEvent, il lui faut
                // une fonction actionPerformed qui sera activée quand le
                // bouton sera cliqué.
            {
                b.setLabel("non");
            }
        });

        setBounds(200,200,100,100);
        setVisible(true);
    }
}

public class Main           // ici classe de test de la classe Fen
                            // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )
{
    public static void main( String [] args)
    {
        Fen mafen = new Fen();
    }
}
```

// 5) c'est la fenêtre qui traite l'événement

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class Fen extends Frame implements ActionListener

    // implémentation de l'interface ActionListener
    // car ma fenêtre est un écouteur d'événements
    // sur les actions sur les contrôles.

{
    private Button b = new Button("oui");

    public Fen()
    {
        add(b);
        b.addActionListener(this);
        // la fenêtre s'enregistre ( this ) comme un écouteur des
        // actions sur le bouton b.
        setBounds(200,200,100,100);
        setVisible(true);
    }

    public void actionPerformed ( ActionEvent evt )

        // la fenêtre étant un écouteur d'ActionEvent, il lui faut
        // une fonction actionPerformed qui sera activée quand le
        // bouton sera cliqué.

    {

        b.setLabel("non");

    }

}

public class Main          // ici classe de test de la classe Fen
                          // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )

{
    public static void main( String [] args)
    {
        Fen mafen = new Fen();

    }

}
```

// 6) c'est l'application (main) qui traite l'événement

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

                             // implémentation de l'interface ActionListener
                             // car mon application est un écouteur d'événements
                             // sur les actions sur les contrôles.
public class Main implements ActionListener
                             // la terminaison de la fenêtre n'est pas traitée ( CTRL-C )
{
    private Button b = new Button("oui");

    public static void main( String [] args)
    {
        Main appli = new Main();
        // mon application est à l'écoute des actions, pour écouter
        // elle doit exister en tant qu'objet pour pouvoir s'enregistrer.

        Frame mafen = new Frame();
        mafen.add(appli.b);
        appli.b.addActionListener(appli);
        // mon application s'enregistre ( appli ) comme un écouteur des
        // actions sur le bouton b.

        mafen.setBounds(200,200,100,100);
        mafen.setVisible(true);
    }

    public void actionPerformed ( ActionEvent evt )
        // l'application étant un écouteur d'ActionEvent, il lui faut
        // une fonction actionPerformed qui sera activée quand le
        // bouton sera cliqué.

    {
        b.setLabel("non");
    }
}
```

La solution 1 est la plus belle, mais elle met en œuvre beaucoup de classes externes, et les écouteurs sont rarement des classes réutilisables. De plus il est souvent nécessaire de gérer des liens avec les composants de l'interface qui seront modifiés par le traitement (exemple du bouton de la solution 1). Cette solution ne sera donc pas pratique quand l'écouteur agit sur l'objet qui veut écouter (la frame ici). Par contre c'est la solution idéale quand l'écouteur est réutilisé.

La solution 2 offre les doubles qualités d'un codage assez simple, et de garder ensemble le codage de la fenêtre et le traitement des événements. C'est la solution adéquate quand l'écouteur ne sera pas réutilisé.

La solution 3 est moins lisible que la 2. La solution 2 lui sera préférée. Pensez que les programmeurs qui vont maintenir votre code seront heureux de trouver le code le plus simple, et le plus explicite possible.

La solution 4 a l'inconvénient d'une faible lisibilité, mais l'avantage d'être très compact. Cette solution devrait être abandonnée. Certains ateliers de développement java proposent de générer automatiquement ce type de code. Je vous conseille de ne pas le faire.

La solution 5 semble simple quand on a à traiter un nombre restreint d'événements, sinon on retombe sur les mêmes problèmes que la solution Java 1.0 (il n'y aura qu'une fonction actionPerformed pour traiter tous les ActionEvents). Il faut savoir que le principe de traitements des événements de java, qui fonctionnait sur un modèle proche de cet exemple a été abandonné car le modèle ne suit pas les préconisations du développement objet. Nous oublierons donc cette approche.

La solution 6 est bien trop déstructurée pour être étendue au traitement d'une application.

Sur chaque exemple rencontré qui ne respecte pas le modèle 1) ou le 2), vous devrez réécrire l'exemple pour qu'il se conforme au modèle 1) ou 2).

3) Liste des événements et des fonctions de traitement.

Voici la liste des événements, des interfaces associées, et les méthodes nécessitées par ces interfaces. Les fonctions d'abonnement à l'écoute d'un événement sont du type addxxx, ou xxx est le nom de l'interface (addActionListener, addMouseListener, ...).

Il est à noter que pour certains événements il y a plusieurs fonctions à implémenter pour réaliser l'interface (jusqu'à sept fonctions pour WindowListener).

Liste des catégories d'événements, des interfaces associées et des méthodes de traitement.

Catégorie	Interface de veille	Méthodes
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
MouseMotion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Key	KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) componentHidden(ComponentEvent) componentResize(ComponentEvent) componentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEvent) windowClosed(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ComponentEvent) componentRemoved(ComponentEvent)
Text	TextListener	textValueChanged(TextEvent)

Liste des événements générés par les composants AWT.

Composant AWT	action	adjustment	component	container	focus	item	key	mouse	mousemotion	text	window
Button	🎵		🎵		🎵		🎵	🎵	🎵		
Canvas			🎵		🎵		🎵	🎵	🎵		
Checkbox			🎵		🎵	🎵	🎵	🎵	🎵		
CheckboxMenuItem						🎵					
Choice			🎵		🎵	🎵	🎵	🎵	🎵		
Component			🎵		🎵		🎵	🎵	🎵		
Container			🎵	🎵	🎵		🎵	🎵	🎵		
Dialog			🎵	🎵	🎵		🎵	🎵	🎵		🎵
Frame			🎵	🎵	🎵		🎵	🎵	🎵		🎵
Label			🎵		🎵		🎵	🎵	🎵		
List	🎵		🎵		🎵	🎵	🎵	🎵	🎵		
MenuItem	🎵										
Panel			🎵	🎵	🎵		🎵	🎵	🎵		
ScrollBar		🎵	🎵		🎵		🎵	🎵	🎵		
ScrollPane			🎵	🎵	🎵		🎵	🎵	🎵		
TextArea			🎵		🎵		🎵	🎵	🎵	🎵	
TextComponent			🎵		🎵		🎵	🎵	🎵	🎵	
TextField	🎵		🎵		🎵		🎵	🎵	🎵	🎵	
Window			🎵	🎵	🎵		🎵	🎵	🎵		🎵

4) Les Adapteurs.

Nous avons vu que la création d'un écouteur nécessite de réécrire toutes les fonctions définies dans l'interface de veille. Pour les interfaces de veille qui n'ont qu'une fonction ce n'est pas grave. Mais pour les interfaces de veille qui ont plusieurs fonctions il faut réécrire le code de toutes ces fonctions même si une seule des fonctions nous intéresse. Par exemple si je veux traiter la fermeture d'une fenêtre par l'icône adéquat, il faut redéfinir les six autres fonctions de l'interface WindowListener avec un corps vide (voir l'exemple 1).

Il a été défini des classes adapteurs (MouseAdapter, WindowAdapter, ...) pour toutes ces interfaces qui ont plus d'une méthode. Ces classes ne comportent que la définition des méthodes de l'interface de veille correspondante, avec un corps vide. Maintenant, plutôt que d'implémenter les interfaces de veille, nous allons hériter des classes adapteurs, en ne redéfinissant que les méthodes dont le comportement nous intéresse (polymorphisme) (voir les exemples 2 et 3). Cela nous évite de définir les fonctions dont le comportement nous indiffère.

// 1) création d'une classe externe qui écoute l'événement fermeture de fenêtre

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class fen extends Frame
{
    public fen()
    {
        ecouteur oreille = new ecouteur();
        // on crée un écouteur pour gérer la fermeture de la fenêtre
        addWindowListener(oreille);
        // l'écouteur s'enregistre ( oreille ) comme un écouteur des
        // actions sur la fermeture de la fenêtre courante.
        setBounds(200,200,100,100);
        setVisible(true);
    }
}

public class ev6            // ici classe de test de la classe fen
{
    public static void main( String [] args)
    {
        fen mafen = new fen();
    }
}

class ecouteur implements WindowListener
    // implémentation de l'interface WindowListener
    // car mon écouteur est un écouteur d'événements
    // sur la gestion des fenêtres.
{
    public void windowClosing ( WindowEvent evt )
        // mon écouteur étant un écouteur de WindowEvent, il lui faut
        // une fonction windowClosing qui sera activée quand la
        // fermeture de la fenêtre sera demandée.
    {
        System.exit(0);
    }

    // ces fonctions sont demandées par l'interface WindowListener
    // et sont donc fournies, même si elles ne font rien (ici!).
    public void windowActivated(WindowEvent evt) {}
    public void windowClosed(WindowEvent evt) {}
    public void windowDeactivated(WindowEvent evt) {}
    public void windowDeiconified(WindowEvent evt) {}
    public void windowIconified(WindowEvent evt) {}
    public void windowOpened(WindowEvent evt) {}
}
```

// 2) création d'une classe externe écoutant l'événement fermeture fenêtre par un adapteur

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class fen extends Frame
{
    public fen()
    {
        ecouteur oreille = new ecouteur();
        // on crée un écouteur pour gérer la fermeture de la fenêtre
        addWindowListener(oreille);
        // l'écouteur s'enregistre ( oreille ) comme un écouteur des
        // actions sur la fermeture de la fenêtre courante.
        setBounds(200,200,100,100);
        setVisible(true);
    }
}

public class ev7            // ici classe de test de la classe fen
{
    public static void main( String [] args)
    {
        fen mafen = new fen();
    }
}

class ecouteur extends WindowAdapter
    // La classe WindowAdapter définit les 7 fonctions de fenêtre,
    // avec un code vide. Il suffit d'en hériter et de redéfinir le
    // comportement désiré pour créer un écouteur.
{
    public void windowClosing ( WindowEvent evt )
        // mon écouteur étant un écouteur de WindowEvent, il lui faut
        // définir le code de la fonction windowClosing qui sera activée
        // quand la fermeture de la fenêtre sera demandée.
    {
        System.exit(0);
    }
}
```

Note : en reprenant les exemples du chapitre 2, vous trouverez facilement comment créer un écouteur, à partir d'un adapteur, en construisant une classe interne, au lieu d'une classe externe.

// 3) création d'une classe anonyme écoutant l'événement fermeture fenêtre par l'adaptateur

```
import java.awt.event.*;    // importation des classes des événements
import java.awt.*;         // importation des classes des composants

class fen extends Frame
{
    public fen()
    {
        addWindowListener(new WindowAdapter()
            {
                // La classe WindowAdapter définit les 7 fonctions de fenêtre,
                // avec un code vide. il suffit d'en hériter et de redéfinir le
                // comportement désiré pour créer un écouteur.
                {
                    public void windowClosing ( WindowEvent evt )
                    // on redéfinit la fonction polymorphique windowClosing pour
                    // lui associer une action à la fermeture de fenêtre.
                    {
                        System.exit(0);
                    }
                });
                // l'écouteur s'enregistre ( anonyme ) comme un écouteur des
                // actions sur la fermeture de la fenêtre courante.
                setBounds(200,200,100,100);
                setVisible(true);
            }
    }
}

public class ev8            // ici classe de test de la classe fen
{
    public static void main( String [] args)
    {
        fen mafen = new fen();
    }
}
```

5) Exercices :

1) première approche.

Les fonctions de traitement des événements ont un paramètre de classe Xevent (X = Window, Mouse, ...). Observez la classe MouseEvent, particulièrement une fonction membre isMetaDown() vraie si le clic est droit et une fonction membre getClickCount qui, comme son nom l'indique compte le nombre de clicks successifs rapprochés. Regardez également les autres méthodes, elles pourraient vous servir.

Nous allons développer une Frame traitant les événements de la souris. Veillez à respecter les 6 étapes proposées (a à f), un certain nombre de propositions vous étant faites pour aller plus loin, vous vous y intéresserez en fonction de votre avancement.

a) Test de la souris

Réaliser une application (pas une applette) qui pour chaque événement souris, écrit sur la console DOS l'action qui a été faite. (bougé pour un événement MouseMoved, pressé pour MousePressed, ...) Vous y inclurez aussi le double clic, et le fait que l'action concerne le bouton droit ou gauche chaque fois que nécessaire. Attention, ce n'est pas aussi simple qu'il n'y paraît : traitez tous les événements, et testez les. Ceci vous permet de bien comprendre le fonctionnement de la souris, et son interprétation par Java.

La méthode isMetaDown() est vraie quand le bouton droit est activé.

La méthode isAltDown() est vraie quand le bouton milieu est activé.

La méthode getButton() vous donne le bouton qui a **changé** d'état (BUTTON1 pour gauche, BUTTON2 pour milieu, BUTTON3 pour droit).

La méthode getModifiersEx() vous donne une radiographie de l'état de la souris. Cela permet de savoir si un ou plusieurs boutons sont appuyés (BUTTON1 _DOWN_MASK ... vous approfondirez).

b) Dessin de lignes

Notre application va tracer des traits en respectant les règles suivantes :

- Bouton gauche pressé : point de départ d'un trait.
- Bouton gauche relâché : point d'arrivée d'un trait.
- Double clic gauche : effacement de tous les traits.
- Clic droit effacement du dernier trait.
- Quand le maximum de traits est atteint, prévenir l'utilisateur dans la barre de titre de la Frame (setTitle).
- Quand on « drag », le trait en cours de construction se trace en couleur différente.

Pour cela vous pourrez vous aider du code suivant, extrait du livre « Apprenez Java 1.1 en 21 jours » , Lemay et Perkins, édition s&sM, p291 et suivantes. Attention ici l'exemple n'est pas développé suivant le modèle objet de traitement des événements. Il vous faudra donc le refaire dans les normes. D'autre part nous voulons ici une Frame, pas une Applette.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Applette extends Applet implements MouseListener, MouseMotionListener
```

```

{
    private final int MAX = 10;           // nombre maximum de traits
    private Point starts[] = new Point[MAX]; // points de départ
    private Point ends [] = new Point [MAX]; // points d'arrivée
    private Point anchor;                 // point de départ de la ligne courante
    private Point currentPoint;           // point d'arrivée de la ligne courante
    private int currline = 0;             // nombre de ligne tracées

    public void init()
    {
        // abonne l'applette aux deux services d'écoute
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // nécessaires pour prétendre être à la fois MouseListener et MouseMotionListener
    public void mouseClicked(MouseEvent me){ }
    public void mouseEntered(MouseEvent me){ }
    public void mouseMoved(MouseEvent arg0) { }
    public void mouseExited(MouseEvent me) { }

    public void mousePressed(MouseEvent me)
    {
        if (currline < MAX)
        {
            anchor = me.getPoint();
        }
        else
        {
            System.out.println("trop de lignes");
        }
    }

    public void mouseReleased(MouseEvent me)
    {
        if (currline < MAX)
        {
            ends[currline] = me.getPoint();
            starts[currline] = anchor;
            currline++;
            currentPoint = null;
            anchor = null;
            repaint();
        }
    }

    public void mouseDragged(MouseEvent me)
    {
        if (currline < MAX)
        {
            currentPoint = me.getPoint();
            repaint();
        }
    }
}

```

```

public void paint(Graphics g)
{
    // dessin des lignes validées
    for (int i=0; i < currline;i++)
    {
        g.drawLine(starts[i].x,starts[i].y,ends[i].x,ends[i].y);
    }
    // dessin de la ligne courante si elle existe
    g.setColor(Color.PINK);
    if ( currentPoint != null )
    {
        g.drawLine(anchor.x,anchor.y,currentPoint.x,currentPoint.y);
    }
}
}

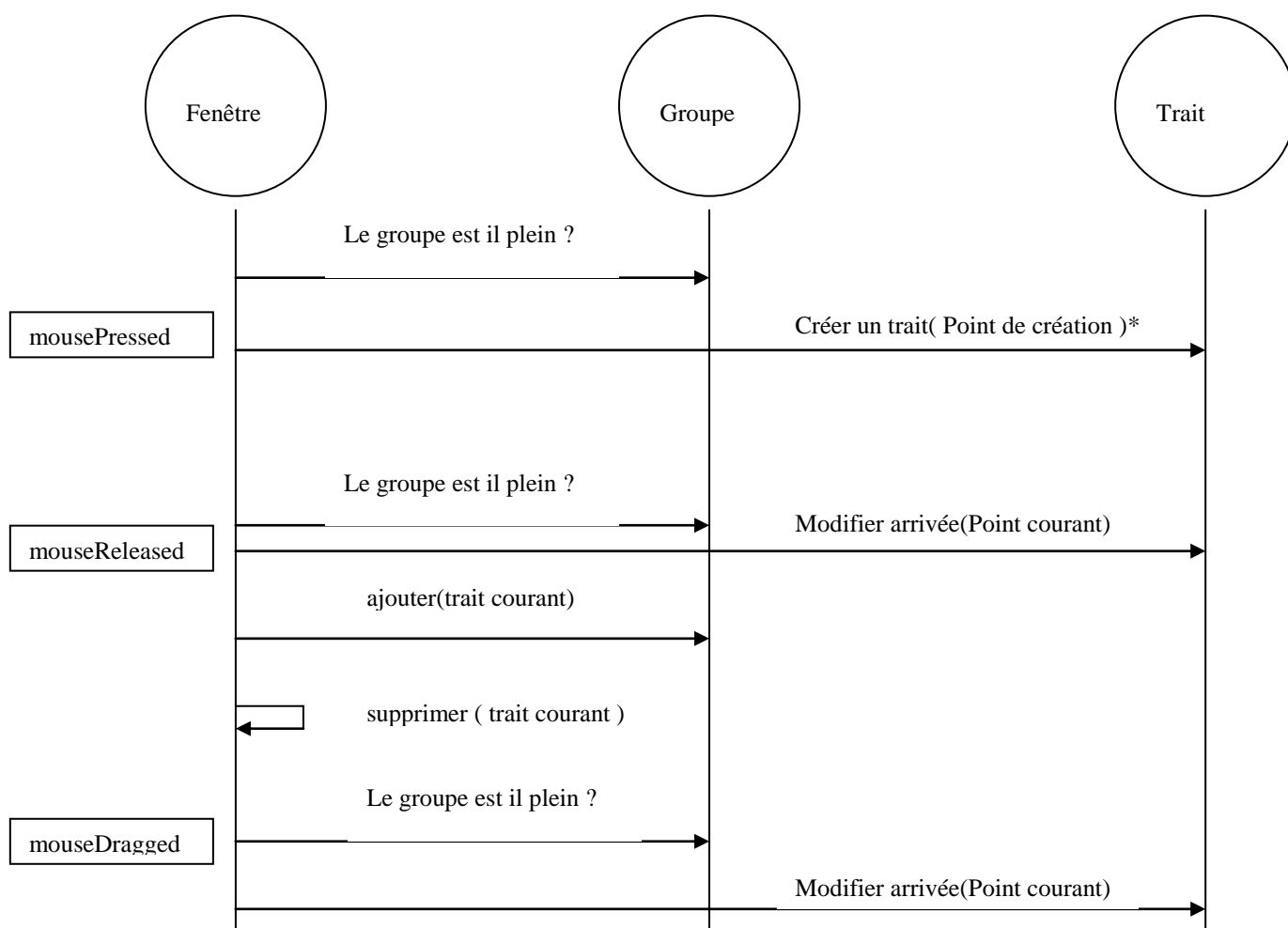
```

Réaliser l'application demandée, en vous inspirant (dans le bon sens) de l'exemple proposé. Vérifiez que votre application est conforme au cahier des charges, vous l'écrirez bien sûr dans les normes de développement demandées. Puis insérer les deux nouvelles fonctionnalités qui ne sont pas traitées dans l'exemple.

c) Conception objet de l'application

Notre dessin est composé d'une collection de traits, et éventuellement d'un trait en cours. Il est étonnant que nous ne retrouvions pas ces notions dans le programme. Notre programme générerait un groupe de traits, et éventuellement un trait en cours, si nous sommes en cours de tracé.

Faisons un petit Diagramme intuitif qui symbolise les échanges entre les classes Fenêtre, Groupe et Trait. Ce schéma nous montre les méthodes nécessaires et suffisantes pour chaque classe.



* ceci ressemble furieusement à un constructeur, ou je ne m'y connais pas...

au MousePressed : création du trait en cours

au MouseDragged : modification du point d'arrivée du trait en cours

au MouseReleased : modification du point d'arrivée du trait en cours, et rangement de ce trait dans la collection. Puis mise à null de ce trait en cours.

Tout ceci uniquement si le groupe de traits n'est pas plein.

Vous **complèterez** ce schéma en particulier sur le mouseClicked, en précisant si droit ou gauche, et si doubleclic, et vous préciserez également ce qu'il faut faire sur la demande de dessin de la fenêtre (paint).

Nous allons développer donc deux classes : la classe Groupe (sûrement proche d'une classe groupe déjà créée), et la classe Trait. Vous étudierez bien, sur chaque événement souris à traiter, les méthodes dont nous avons besoin dans ces deux classes (grâce au schéma intuitif précédent).

Cela nous donne une vraie application objet. La classe fenêtre est plus lisible, les traitements propres aux traits, et ceux propres aux groupes, ont été traités séparément, dans leur classe respective.

Nous en profiterons aussi pour donner une couleur aléatoire aux traits à leur création. Cette couleur n'étant jamais modifiée au cours de la vie du trait.

Vous générerez les couleurs aléatoirement par l'instruction :
`new Color((int)(Math.random()*0xFFFFFF))`

d) Nouveau trait

Pour nous assurer que nous avons bien travaillé :

Un Trait maintenant est défini comme suit :

- le point de départ est le centre d'un cercle.
- le point d'arrivée nous définit un point du cercle.
- une couleur sera générée aléatoirement pour tracer un cercle plein.

Le rayon du cercle se calcule suivant la formule : $r = \text{racine carrée } ((x1-x2)^2 + (y1-y2)^2)$.

Le cercle plein se tracera avec la méthode « `fillOval` » de la classe `Graphics`.

Extrait de la documentation de cette méthode :

```
public abstract void fillOval(int x,  
                               int y,  
                               int width,  
                               int height)
```

Fills an oval bounded by the specified rectangle with the current color.

Parameters:

- `x` - the `x` coordinate of the upper left corner of the oval to be filled.
- `y` - the `y` coordinate of the upper left corner of the oval to be filled.
- `width` - the width of the oval to be filled.
- `height` - the height of the oval to be filled.

Réaliser l'application demandée, en ne modifiant que la méthode `paint` de la classe `Trait`.

Pour aller plus loin :

Antialiasing, ou comment obtenir un contour d'aplatissement sans « crénelage »:

Lors du remplissage de formes courbes, on peut atténuer l'effet de crénelage en utilisant une des méthodes de rendu de la classe `Graphics2D` (le système calcule des couleurs intermédiaires pour la transition entre 2 couleurs, rendant ainsi le contour plus net pour l'œil).

```
public void paint(Graphics g){  
    Graphics2D gd = (Graphics2D)g;  
    gd.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);  
    gd.fillOval(10, 10, 30, 60);  
}
```

Ligne 1 : récupération du `Graphics2D` pour le contexte en cours.

Ligne 2 : demande au système de dessiner en utilisant l'anti-aliasing

Ligne 3 : dessin d'un ovale

Amélioration de l'affichage (diminuer le clignotement)

Chaque fois que l'on trace un trait, il se produit un clignotement fatigant pour l'œil. Pour éviter cela vous pouvez utiliser le double buffering, qui consiste à effectuer le tracer en mémoire, puis à basculer sur le tracé mémoire une fois que celui-ci est complètement effectué.

Exemple de code :

Image brouillon ;
Graphics2D tampon ;

```
public void paint( Graphics g)
{
    Image brouillon ;
    Graphics2D tampon ;

    Dimension d=this.getSize();

    brouillon = createImage(d.width,d.height);
    tampon = (Graphics2D)brouillon.getGraphics();
    // ici faire tout le boulot d'affichage
    tampon.setColor(Color.black);
    tampon.drawString( "bonjour", 10, 45);
    // maintenant faisons une image du brouillon
    g.drawImage(brouillon,0,0,null);    // affichage effectif
}
```

Il faut également redéfinir la méthode update pour que cela fonctionne bien :

```
Public void update(Graphics g)
{
    paint(g) ;
}
```

e) Nouveau type de trait

Pour vous convaincre de l'excellence de notre solution par rapport à celle du livre nous allons introduire une petite variante à notre application : Les traits que nous allons tracer ne sont plus définis par le point de départ et le point d'arrivée, mais par l'ensemble des points suivis par le curseur de la souris.

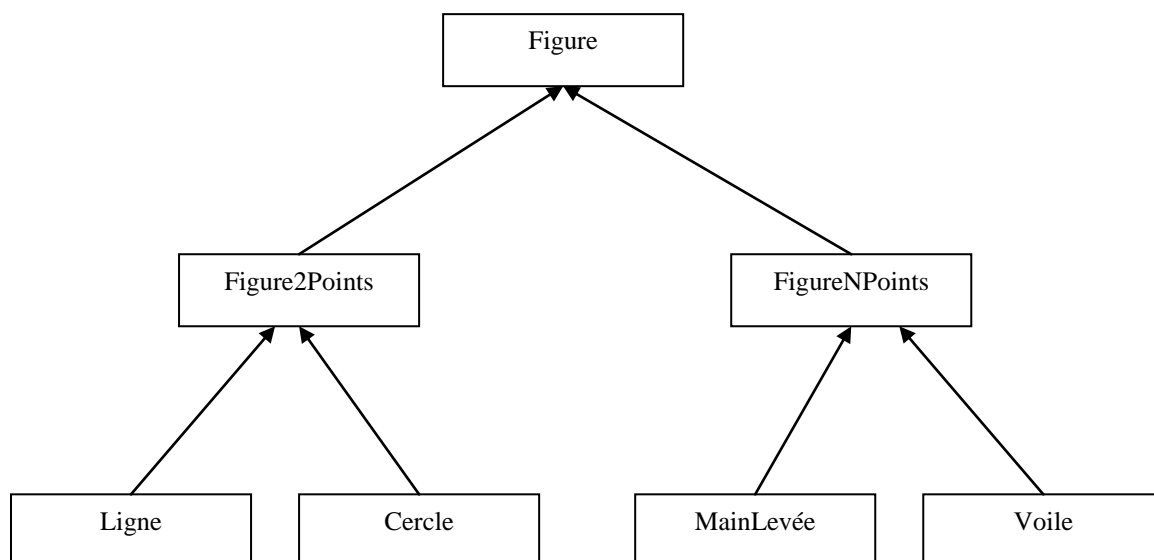
Que faire pour traiter ce problème ? Clairement les événements souris sont les mêmes. Donc la classe Fenêtre ne sera pas modifiée. La gestion des Groupes ne sera non plus pas touchée. Nous n'avons qu'à modifier la classe Trait pour arriver au résultat. Dans cette classe les interfaces des méthodes ne seront pas modifiées, en effet les informations en entrée ne pourront pas être autre chose que ce qu'elles sont, et les services (interface ou contrat ou protocole) de l'objet trait n'ont pas à être modifiés. Par contre il va falloir modifier le contenu de la classe Trait (sa partie privée, soit ses attributs, et le code de ses méthodes).

Un Trait devient un Vecteur de Points, Le constructeur de Trait mettant un Point dans le Vecteur, la méthode **modifierArrivée** rajoutant un autre point au Vecteur. La méthode **paint** affichant les traits entre tous les points du Vecteur (attention, si il n'y a qu'un point, de tracer un point). En quelques minutes nous avons transformé notre application de manière très simple et très propre.

f) restructuration de notre application

Nous allons définir un nouveau type de trait : nous gardons la mémoire de tous les points par lesquels nous sommes passés, le tracé consiste à relier le point de départ à tous ces points. Cela formera comme un voile.

Nous avons 4 sortes de traits. Nous pouvons remarquer que si tous sont des traits, les cercles et les lignes ne diffèrent entre eux que par la méthode paint, ainsi que la mainlevée et le voile. Nous allons définir une hiérarchie de classes comme suit :



Les classes **Figure**, **Figure2Points** et **FigureNPoints** sont des classes qui ne servent qu'à la classification. Ce sont des classes abstraites (abstract class **Figure**...).

Les classes terminales sont des classes concrètes, ce sont les traits qui constituent mon dessin.

Nous allons lister ce que nous devons trouver dans chaque classe (ici je ne suis pas exhaustif, seulement indicatif : vous devrez compléter ce que je propose avec ce qu'il faut pour que chaque trait soit opérationnel).

Les classes terminales :

Elles ne diffèrent d'entre elles que par la méthode paint. Elles contiennent donc la méthode paint.

Les classes **FigureXpoints** :

Classes abstraites.

Elles contiennent les données permettant de conserver les informations sur les coordonnées du trait (deux points, ou un vecteur de points). Ces données seront protégées (public pour celui qui hérite car il a besoin des données pour dessiner) et privé pour les autres.

Elles définissent les méthodes permettant d'initialiser le point de départ, et de modifier le point d'arrivée.

La classe Figure :

Classe abstraite

Elle contient l'information de couleur du trait (privée).

Elle contient un constructeur d'initialisation (à partir du point de création). Ce constructeur initialise la couleur aléatoirement. Si le point de départ est nul, l'initialise en (0,0). Puis initialise le point de départ et modifie le point d'arrivée.

Elle contient deux méthodes abstraites : une protégée pour initialiser le point de départ, et l'autre publique pour modifier le point d'arrivée. Ces méthodes sont abstraites car dans cette classe nous ne savons pas comment initialiser le point de départ, ou modifier le point d'arrivée.

Elle contient aussi la méthode paint qui positionne la couleur de tracé à la bonne couleur, le reste du tracé étant fait dans les classes concrètes (qui font donc un super.paint()).

En faisant cela vous constaterez que les classes Cercle et Ligne fonctionnent correctement, alors que les classes MainLevée et Voile génèrent une NullPointerException. Vous utilisez le vecteur alors qu'il n'est pas encore créé.

Voilà l'ordre dans lequel les initialisations sont faites en java à l'appel d'un constructeur :

- 1) Les données membres de la classe fille sont toutes mises à zéro binaire (null pour les références, false pour les booléens, ...).
- 2) Le constructeur de la classe mère est appelé.
- 3) Les données membres sont initialisées dans la classe fille.
- 4) Les instructions suivant l'appel du constructeur de la classe mère sont exécutées.

En déduire où initialiser le vecteur.

Nous désirons maintenant que notre application permettent de mixer sur un même dessin tous ces traits.

Notre groupe devient un groupe de Figures.

Nous mettrons dans la Frame un écouteur de clavier, qui pour chaque touche associe un caractère v pour voile, c pour cercle, m pour MainLevée, et ligne pour le reste. Ce caractère est initialisé à l (pour ligne).

Au moment de la création du trait, il suffit de créer le bon trait en fonction du caractère (un switch).

Notre application offre maintenant de plus intéressantes alternatives. Pour associer un nouveau type de trait, cela devient très rapide.

Pour les plus téméraires :

Le but est de pouvoir ajouter un nouveau type de trait sans toucher à une ligne du code de la Frame.

Le type de trait est conservé dans un type énuméré. Chaque nom de classe commencera par une lettre différente pour les différents traits.

Quand on ajoute une classe il suffit de rajouter le nom de la classe dans le type énuméré.

Quand on frappe au clavier il suffit de chercher la bonne valeur dans le type énuméré, ou la valeur par défaut.

Quand on crée un trait, il faut appeler le constructeur dont le nom est le même que le type énuméré. Voici le code permettant de le faire.

Appel dynamique de constructeur (ici à partir de type énuméré) :

On crée un type énuméré comprenant toutes les valeurs des classes dont on a besoin et une variable de ce type.

```
public enum Forme{Cercle,Ligne,Courbe,Voile};
private Forme _figure;
```

Initialisation:

On initialise la variable de type Forme.

```
_figure = Forme.Ligne;
```

Mise à jour:

On assigne une valeur à _figure selon le besoin

```
_figure = Forme.Cercle; ou _figure = Forme.Courbe; ou _figure =
Forme.Voile ou _figure = Forme.Ligne
```

Appel dynamique du constructeur:

```
try {
    Class<?> [] tab = {Point.class};
    // liste des types des parametres
    Object[] param = {e.getPoint()};
    // liste des parametres
    // si les formes sont dans un package, il faut rajouter le nom du
    // package au nom de classe Figures.Cercle
    String pack;
    if (Cercle.class.getPackage()== null)
    { // pas de package
        pack = new String() ;
    }
    else
    { // un package: le nom de la classe contient le nom du package
        String pack = Cercle.class.getPackage().getName()+ '.' ;
    }
    traitCourant=(Trait)Class.forName(pack + _figure.toString())
        .getConstructor(tab).newInstance(param);
} catch (Exception e1) {
    System.out.println("ça merde !!!");
    e1.printStackTrace();
}
```

- 1) `_figure.toString()` : renvoie la chaîne correspondant à une des valeurs de Figure
- 2) `Class.forName(_figure.toString())` : renvoie la classe de nom "`_figure.toString()`"
- 3) `.getConstructor(tab)` : renvoi le constructeur ayant le tableau `tab` d'arguments (dans notre cas un `Point`)
- 4) `.newInstance(param)` : fabrique une instance avec `param` comme argument pour le constructeur "`param`" (dans notre cas le point cliqué)
- 5) `traitCourant` est de type "`Trait`", il faut donc caster l'instance créée en "`Trait`".
- 6) `pack` est le nom du package suivi d'un point si il y a un package

Avantage:

L'ajout de nouvelle valeur dans le type énuméré ne modifie pas l'appel dynamique au constructeur.

Ici nous avons utilisé les capacités d'introspection des objets qui permettent de demander à un objet de quoi il est constitué, et de lui demander des services de manière plus puissante mais aussi un peu plus compliquée. A ne réserver que pour des traitements particuliers.

Pour les maniaques :

Nous commençons à avoir un certain nombre de classes, il est peut être temps de mettre un peu d'ordre dans ce bazar.

Nous allons créer des packages (sous ensembles de classes cohérentes) par exemple pour tout ce qui concerne les traits.

Nous allons aussi constituer des ensembles livrables des jars (soit des applications, soit des bibliothèques de classes que l'on peut utiliser dans un projet).

Quelques notions sur le package et le .jar

Package

Définition d'un package :

Un package est un ensemble de classes qui se situent sous un même répertoire.

Les classes peuvent être rangées dans divers répertoires et sous répertoires.

Le nom du package est composé des noms des répertoires formant le chemin d'accès à ces classes, séparés par un '`.`'.

Déclaration d'un package :

Une classe appartenant à un package a pour première instruction :
`package MonRep.MonSousRep;`

Utilisation d'un package :

On peut utiliser des packages pour deux raisons.

Pour organiser un projet (en répertoire, sous répertoire).

Pour mettre ensemble des classes qui ajoute des fonctionnalités au langage (par exemple la classe `Entier` ou `Lire`, que l'on utilise partout dans les exercices).

- ✍ Pour utiliser un package dans un autre projet, il ne faut pas oublier l'instruction « *import MonRep.MonSousRep;* »
- ☛ Une méthode « *main(String []arg)* » ne doit pas être dans une classe qui appartient à un package.

JAR (Java **AR**chive)

✍ Définition d'un JAR

Un JAR est une archive (fichier unique) compressé ou non de classes java et des ressources.

Créer un package

A partir d'un projet existant

- ⇒ Sélectionner le projet contenant les classes à mettre dans le package. Ces classes sont sous le « *default package* ».
- ⇒ Clic droit sur le projet > **New** > **Package**.
- ⇒ Donner un nom au package ⇒ Eclipse crée un répertoire de même nom.
- ⇒ Faire glisser les classes, qui vont constituer le package, du « *default package* » vers le package que vous venez de créer.

Créer un package

Créer un nouveau projet **File** > **New** > **Project**.

Clic droit sur le projet > **New** > **Package**.

- ⇒ Donner un nom au package ⇒ Eclipse crée un répertoire de même nom.
- ⇒ Pour insérer une classe dans ce package 2 méthodes :
 - ⇒ Clic droit sur le projet > **New** > **class** et dans la boîte de dialogue « new java class » mettre le nom du package auquel appartient cette nouvelle classe.
 - ⇒ Clic droit sur le package > **New** > **class**.

Créer un .jar

- ⇒ Clic droit sur le projet > export > JAR file, bouton Next => on tombe sur l'assistant qui nous permet de choisir :
 - Le nom du fichier JAR
 - Sélectionner les classes à inclure dans le JAR

Ajouter un jar dans un projet

Au moment de la création du projet

- ⇒ File > New > Project
- ⇒ Sélectionner Java Project , bouton Next
- ⇒ Choisir le nom du projet, bouton Next

- ↳ Se positionner sur l'onglet Libraries
 - ⇒ Si le JAR est dans le workspace
Utiliser le bouton Add JARs et sélectionner le(s) fichier(s).
 - ⇒ Si le JAR n'est pas dans le workspace
Utiliser le bouton Add External JARs et sélectionner le(s) fichier(s).

A tout moment après la création d'un projet

- ↳ Clic droit sur le Projet > Propriétés
- ↳ Cliquez sur « Java Build Path »
- ↳ Sélectionner l'onglet « Libraries »
 - ⇒ Si le JAR est dans le workspace
Utiliser le bouton Add JARs et sélectionner le(s) fichier(s).
 - ⇒ Si le JAR n'est pas dans le workspace
Utiliser le bouton Add External JARs et sélectionner le(s) fichier(s).

Pour finir, juste pour le fun je vous propose de mettre en place une fabrique, et un décorateur :

Dans un premier temps nous allons regrouper tout ce qui concerne la fabrication des figures dans une seule classe qui ne sert qu'à ça : une fabrique simple (pattern). Donc le type de figure courant y est, ses fonctions d'accès, et la création d'une figure. Ainsi à chaque nouvelle figure ajoutée au logiciel, nous n'avons qu'à modifier la classe fabrique simple. Cette classe pourra éventuellement être accessible par un singleton (pattern).

Dans un deuxième temps il me vient l'idée de créer des figures avec une ombre (voire avec une ombre double...). Mon dessin pourrait alors avoir des figures sans ombre, des figures ombrées, ou avec une ombre double, et ce pour n'importe quel type de figure. Pour cela je vous propose de mettre en place un décorateur (pattern). Dans un premier temps construisez une interface (Fantome) pour les figures. Le Dessin contiendra alors un vecteur de cette interface Fantome. Un décorateur de figure est un Fantome qui contient une figure. Une FigureOmbrée est un décorateur de figure qui contient une deuxième Figure identique à la précédente mais un peu décalée, et un peu plus sombre. A noter que le décorateur se construira avec un Fantome en paramètre ce qui permet de mettre une ombre double si on veut. Enfin l'interface permet de mettre en commun des Fantomes construits à partir des points (les Figures) et des Fantomes construits à partir d'autres Fantomes, c'est-à-dire des figures ombrées, ou doublement ombrées. Nous pourrions imaginer d'autres décorateurs. Le tout se retrouve mixé dans notre dessin. Dans les exemples fournis vous trouverez un fichier decorateur.jar auto exécutable qui vous montre l'application obtenue.

2) création d'interface graphique sans souci de disposition sur l'écran.

Une fenêtre (Frame) est une zone d'affichage dans laquelle nous allons pouvoir disposer des contrôles, ou d'autres zones d'affichage.

Pour créer un contrôle dans une zone d'affichage, il suffit de le créer, puis de l'ajouter à la zone d'affichage.

```
Frame f = new Frame() ;  
Label l = new Label("coucou");  
f.add(l);
```

Si la fenêtre possède un comportement non standard des fenêtres (peut être parce qu'il écoute un bouton), il sera alors préférable de définir une nouvelle classe fenêtre qui hérite de Frame.

```
public class maFrame extends Frame  
{  
    private Label l = new Label ("coucou");  
    public maFrame ()  
    {  
        super();  
        add(l);  
    }  
    public void nouveaucomportement()  
    {  
  
    }  
}
```

Une fenêtre contient une zone d'affichage. Il arrive que cette zone d'affichage soit subdivisée en sous panneaux d'affichage (Panel). Chacun de ces sous panneaux peut alors être géré indépendamment.

Quand nous voudrons traiter des événements, trois cas se présenteront à nous :

- Nous voulons nous écouter nous même (une fenêtre écoute les événements souris sur elle-même, une fenêtre écoute sa croix de fermeture, ...).
- Nous voulons écouter un de nos composants (une fenêtre écoute son bouton, ...).
- Nous voulons écouter un composant que nous ne connaissons pas. C'est possible si nous connaissons quelqu'un qui le connaît, ou quelqu'un qui connaît quelqu'un qui le connaît, ... (une fenêtre écoute un bouton qui est dans un sous panel de la fenêtre, une fenêtre écoute un bouton d'une autre fenêtre, ...).

Nous vous proposons un exercice où nous mettrons en évidence ces trois manières possibles d'écouter des composants. Cela nous permettra de mettre en place une écoute d'événement dans tous les cas possibles.

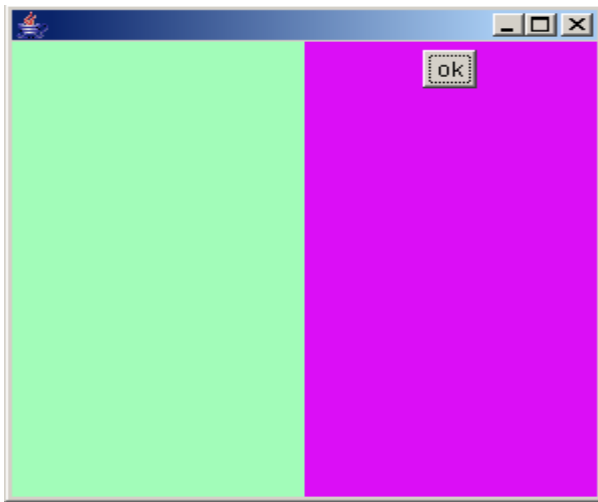
Soit une fenêtre qui partage sa zone d'affichage en deux parties (ici c'est pour les besoins de ce que je veux vous faire découvrir : une fenêtre n'a pas toujours besoin de panel, car elle dispose d'une zone d'affichage). La première est un panel qui restera désespérément vide, mais qui prendra des couleurs. La deuxième est un panel qui contient un bouton ok.

Pour le premier panel, nous n'ajoutons rien de particulier, c'est donc une instance de la classe Panel. La deuxième c'est un panel qui a un comportement, et un attribut propre (le bouton). Nous redéfinirons donc une classe héritant de Panel pour décrire cette nouvelle classe (Pane par exemple).

La fenêtre définit deux attributs, le Panel, et le Pane, puis les ajoute à sa zone d'affichage (add) après avoir défini que l'on veut mettre les panels sur deux colonnes (`setLayout(new GridLayout(1,2,10,10) ;`). Nous approfondirons ultérieurement la logique d'affichage.

Nous voulons :

- Que quand le bouton ok est appuyé, le Panel prenne une couleur aléatoire (ici c'est la fenêtre qui écoute le bouton).
- Que quand le bouton ok est appuyé, le Pane prenne une couleur aléatoire (ici c'est le Pane qui écoute le bouton).
- Vous traiterez ensuite l'événement de fermeture de la fenêtre par la petite croix.



Vous générerez les couleurs aléatoirement par l'instruction :
`new Color((int)(Math.random()*0xFFFFFFFF))`

3) la disposition (ou layout) et l'encadrement (ou insets)

le layout est la manière de disposer les différents contrôles sur une surface adéquate.

Le layout par défaut est le flowlayout qui dispose les contrôles les uns à la suite des autres, comme il le peut.

Le borderlayout est une disposition de type nord, sud, ouest, est, centre. Chacune de ces zones pouvant être elle-même une zone redécomposable.

Le gridlayout est une disposition en forme de tableau.

Le gridbaglayout est la disposition qui permet de mettre ce qu'on veut, où on veut, mais hélas c'est la moins simple. C'est la disposition utilisée par les générateurs d'interface, si vous voulez l'utiliser manuellement cela demande un peu d'investissement.

Le GroupLayout est le layout de référence pour beaucoup d'IDE depuis la version 6, et à tendance à se généraliser.

Les autres layouts sont d'un abord simple dans tous les documents et seront utilisés de préférence quand on doit construire une interface à la main.

Les insets sont les marges qui sont laissées sur un conteneur .

Pour définir un Insets il suffit de redéfinir (polymorphisme) la fonction getInsets du conteneur.

```
public Insets getInsets ()
{
    return new Insets(10,20,30,40) ;
}
```

Le conteneur a ainsi une marge droite de 40 pixels, gauche de 20 pixels, haute de 10 pixels, basse de 30 pixels.

a) fenêtres communicantes

Construire une fenêtre contenant deux boutons ouvrir et fermer.

Ouvrir permet d'ouvrir une fenêtre.

Fermer permet de cacher la ou les fenêtres ouvertes (hors la fenêtre principale).



La fenêtre affiche un label et contient un bouton ouvrir boîte de dialogue.

Le bouton ouvrir boîte de dialogue permet d'ouvrir une boîte de dialogue (c'est aussi une fenêtre).

La boîte de dialogue a un champ de saisie et un bouton OK.

Le bouton OK permet d'afficher le texte du champ de saisie dans le label de la fenêtre.

Bien sûr vous utiliserez un gestionnaire de disposition (layoutmanager), et des cadres pour les conteneurs.

Vous programmerez également les fermetures de fenêtres comme suit :

- fermeture de la fenêtre principale : arrêt de l'application.
- fermeture de la fenêtre : cacher la fenêtre ainsi que la boîte de dialogue si nécessaire.
- fermeture de la boîte de dialogue : cacher la boîte de dialogue.

b) nous allons enfin mettre un menu, et faire communiquer deux fenêtres.

Regarder dans les livres ou documents comment créer des menus, et comment réagir au choix de menu (Menu, MenuBar, MenuItem, setMenuBar ...).

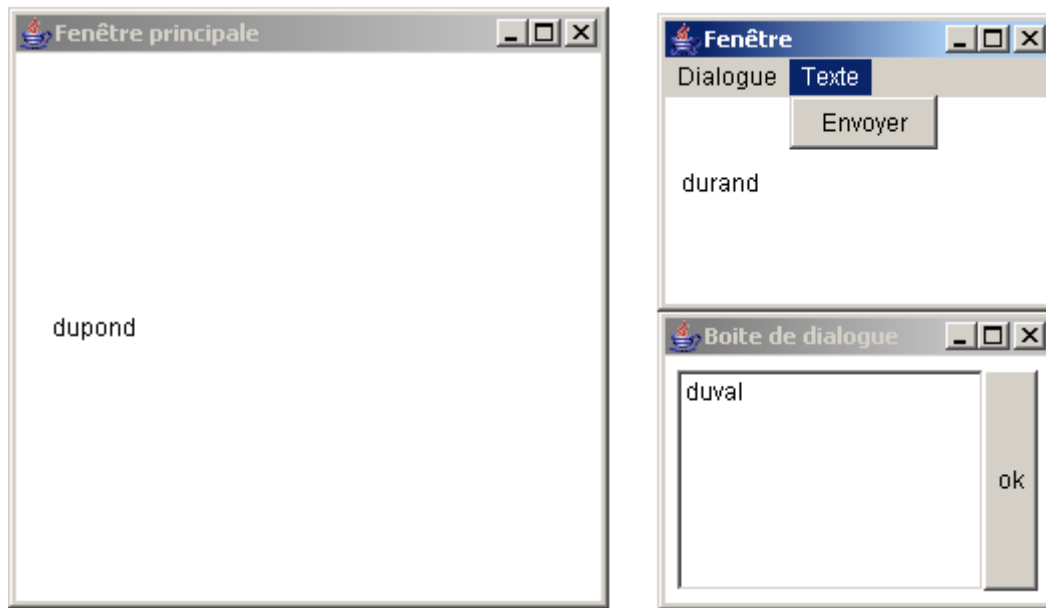
Exercice :

Soit une fenêtre principale avec un label. Cette fenêtre crée une autre fenêtre.

La fenêtre à un menu dialogue avec deux sous-menus ouvrir et fermer.

La fenêtre à un menu texte avec un sous menu envoyer.

La fenêtre à aussi un label.



Ouvrir ouvre une boîte de dialogue identique à celle de l'exercice précédent. Elle permet de saisir un texte, et quand son bouton OK est activé, elle met à jour le label de la fenêtre.

Fermer cache la boîte de dialogue.

Envoyer permet de mettre à jour le label de la fenêtre principale.

Vous noterez que la dernière fenêtre est identique à la dernière fenêtre de l'exercice précédent. Donc, si nous avons bien fait notre travail, nous pouvons reprendre, sans y toucher, la classe de l'application précédente. Cela nous prouvera, si nous n'intervenons pas sur cette classe, que notre conception (découpage du travail entre les classes, répartition des rôles) était bien faite.

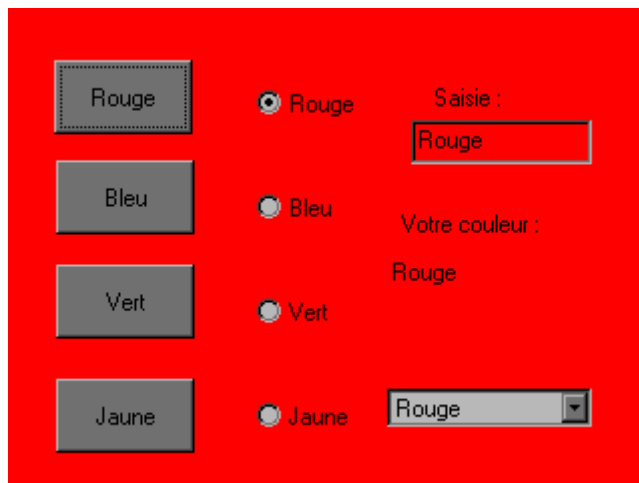
4) gestion d'une interface fenêtrée, avec interaction d'un composant sur un autre

Nous désirons réaliser une boîte de dialogue qui change de couleur de fond. Nous nous limiterons à quatre couleurs : rouge, vert, bleu et jaune. Pour changer la couleur de fond nous pourrions

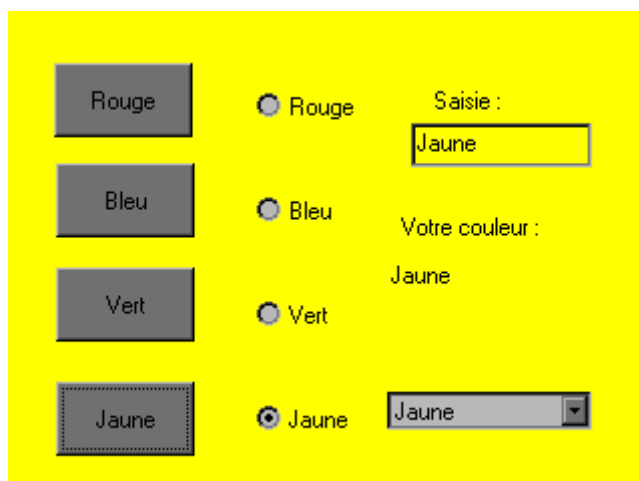
- Cliquer un bouton de la couleur du fond.
- Appuyer sur un radio bouton.
- Saisir la couleur de fond
- Cliquer dans une liste déroulante.

Bien entendu, quelque soit la manière de changer la couleur du fond, les autres moyens sont mis à jour.

Soit l'interface suivante:



Que l'on appuie sur le bouton jaune, que l'on appuie sur le bouton radio jaune, que l'on saisisse jaune (en majuscule ou en minuscule), ou que l'on choisisse jaune dans la liste déroulante, on obtient le résultat suivant:



Il est important ici de définir une stratégie globale, qui permette de résoudre le plus simplement possible ce problème.

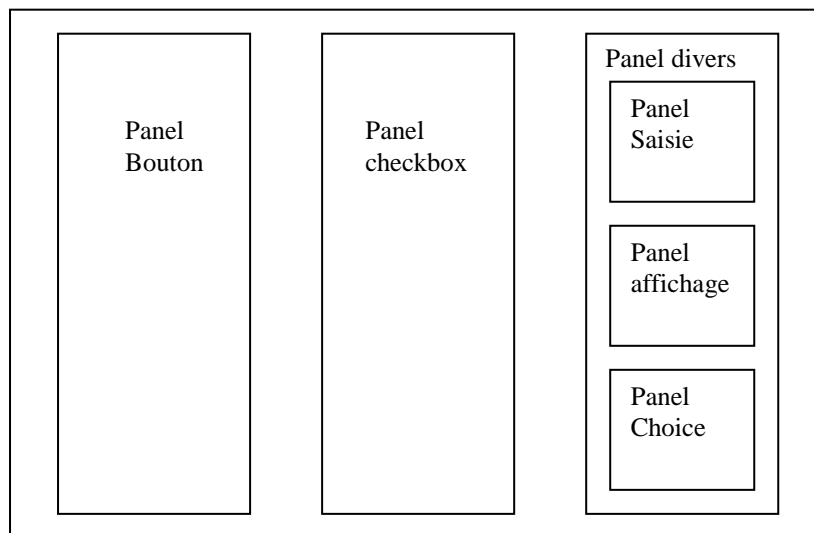
Ici nous vous proposons une stratégie de conception du problème, dans la logique de raisonnement des exercices précédents. Dans le chapitre 3 nous prendrons une autre philosophie de résolution de ce problème.

Dans ce problème nous nous poserons cinq questions :

- 1) Comment construire cette interface ?
- 2) Quel outil pour remettre à jour la fenêtre (ou repeindre) ?
- 3) Qui écoute les différents contrôles ?
- 4) Combien d'écouteurs pour gérer ces contrôles ?
- 5) Voit-on des outils communs ?

Voici les réponses que je vous conseille :

- 1) Nous allons gérer cette interface avec un certain nombre de panels, pour pouvoir disposer les composants comme le désire le client, et pour regrouper dans des panels différents les différents contrôles. Ainsi chaque panel traite sa logique.



- 2) Pour peindre nous allons utiliser la méthode setBackground. Ici setBackground ne fera pas que peindre la couleur de fond, elle demandera aussi aux composants imbriqués de peindre leur couleur de fond, et fera les mises à jour nécessaires (comme le focus sur le bon bouton (requestFocus()), sélectionner le bon Checkbox, ...). La Fenêtre principale, et chaque Panel devra donc faire un polymorphisme de la méthode setBackground.
- 3) Celui qui écoute doit connaître tous les panels, afin de leur demander de peindre. Or, un seul objet connaît tous les autres : c'est la fenêtre. Ici ce sera donc la fenêtre qui écoutera tous les contrôles.
- 4) D'un point de vue fonctionnel, le travail est le même que l'on active n'importe quelle sorte de contrôle. Il est donc logique d'avoir un seul écouteur. Cet écouteur devra implémenter deux types différents d'interface, du fait de la multiplicité des types de contrôle.

- 5) Nous constatons que lorsque nous peignons certains panels, nous avons besoin certes de la couleur, mais aussi de la chaîne de caractères correspondant à cette couleur. Il serait bon d'avoir une méthode qui convertisse une couleur en chaîne de caractères. Quand nous actionnons un contrôle, nous récupérons dans tous les cas une chaîne de caractères (il faudra se débrouiller pour cela) qui représente le nom de la couleur. Il nous faut donc une méthode qui convertisse cette chaîne de caractères en couleur. Je vous propose de créer deux outils, définis dans une boîte à outils, ayant les interfaces suivantes :

```
String colorToString( Color c ) ;  
Color stringToColor( String s ) ;
```

Pour les esthètes:

Il pourrait être intéressant de créer une collection de couleur (couple couleur et nom de la couleur) par exemple dans une HashTable. Cela permet de varier les couleurs, tant par les teintes et leur nom que par le nombre de couleurs proposées. A la création de l'interface il suffit de parcourir la collection et de créer autant de composants qu'il y a de couleurs. Toute modification de la collection se répercute automatiquement lors de la prochaine exécution de notre application, sans changer une ligne de code.

5) Les fichiers

1) examen de la classe File

La classe file permet de créer des objets fichiers, mais aussi d'obtenir toute sorte d'information sur les fichiers.

Réaliser une application (pourquoi ne peut-on pas réaliser une applet ?) qui liste dans une liste déroulante (List) les fichiers d'un répertoire (le vôtre par défaut).

Pour un fichier sélectionné :

- Si c'est un répertoire on liste dans la liste déroulante les fichiers du répertoire.
- Si c'est un fichier classique, on affiche dans une fenêtre quelques caractéristiques du fichier parmi nom, répertoire, droits en écriture, en lecture, date de dernière modification (en français bien sûr)...
- Possibilité de remonter dans l'arborescence des fichiers. (optionnel)

2) examen des classes InputStream, FileInputStream, DataInputStream, BufferedInputStream (**exercice optionnel**)

Nous n'évoquerons, ici, que les entrées. Tout ce qui est dit sur les entrées est valable pour les sorties (output).

La classe InputStream est la classe de base des entrées. System.in est un InputStream. Les premiers exercices nous ont montré qu'on n'en faisait pas grand chose sans enrichir cette classe.

La classe FileInputStream permet de définir un flux à partir d'un fichier (voir les constructeurs). Nous ne pouvons toujours lire qu'un flot d'octets.

La classe BufferedInputStream permet de faire des entrées bufférisées. Elle permet de charger un tampon, qui sera vidé petit à petit par les différentes lectures.

La classe DataInputStream permet de lire des données de tout type. Attention toutefois aux fins de ligne.

Ces classes se cascaded. Nous construirons un DataInputStream à partir d'un InputStream (un FileInputStream est un InputStream).

Regarder ces classes et particulièrement les héritages et les constructeurs pour comprendre cette mécanique.

Faire un fichier contenant la liste des fichiers d'un répertoire, et quelques attributs du fichier (répertoire ou fichier, date de dernière modification,...).

3) Sérialisation

La sérialisation permet à un objet de se sauvegarder, ou de se restaurer. En java il suffit pour cela de déclarer que la classe implémente l'interface Serializable. Java se charge de fournir le code qui permet d'écrire un objet, ou de lire un objet (writeObject ou readObject).

Exemple :

Class Truc implements java.io.Serializable

```
{
    int x ;
    String y ;
```



```

    Truc( int a, String b)
    {
        x = a ;
        y = b ;
    }
}

```

La classe Truc est sérialisable.

```

Truc machin = new Truc(10, "coucou");
FileOutputStream fos = new FileOutputStream("monfichier.txt") ;
ObjectOutputStream oos = new ObjectOuputStream( fos ) ;

oos.writeObject( machin) ;
oos.close() ;

```

Il est possible de définir que des données membres d'une classe ne doivent pas être sauvegardées. Le mot-clef « transient » a cet usage. Par la suite nous en verrons un exemple d'utilisation.

Le comportement de sauvegarde d'un objet est défini par défaut par java. Si ce comportement ne vous satisfait pas (en particulier dans le cas d'un héritage, et de relation particulière entre la classe fille et la classe mère) il est possible de redéfinir les comportements de sauvegarde des objets d'une classe. La classe est alors externalisable (elle implémente Externalizable). Soyez très prudent lors de la redéfinition de ces fonctions.

Reprenons notre exercice de supervision de locaux sensibles.

Nous voulons qu'un fichier « journal.txt » contienne tous les messages reçus par notre supervision, ainsi que toutes les alarmes.

Nous allons reprendre notre programme testant l'historique des alarmes et des messages. Quand notre programme démarre, il charge l'historique qui a été sauvegardé dans un fichier (la première fois le fichier n'existe pas, l'historique sera donc vide). Puis le programme rajoute ses messages et alarmes, et enfin sauvegarde le tout dans un fichier.

L'historique aura donc deux nouvelles méthodes :

- public static Historique load(String nomfichier) ;
- public void save(String nomfichier);

La méthode load est statique, car c'est bien un comportement de l'Historique, mais load crée un historique. Nous ne disposons donc pas d'objet historique pour lui demander de se charger.

Développez, testez et documentez votre nouvelle application.

6) Synthèse sur l'interface et le traitement des événements.

Nous voulons développer une simulation de supervision, avec dans notre fenêtre deux panels. Le premier nous permettra de simuler la création d'un message ou d'une alarme. La deuxième contiendra une liste des différents messages ou alarmes. La liste contient pour chaque message un numéro et le libellé du message. Un double click sur un élément de la liste permet de visionner le contenu complet du message. Pour une alarme le double click permet de visionner le contenu complet du message, mais aussi de le prendre en compte ou de le solder.

La liste est initialisée au démarrage de l'application grâce à la sauvegarde de l'historique, et elle est mise à jour chaque fois qu'une nouvelle alarme ou un nouveau message est créé.

Exemple d'interface :

The interface consists of two windows. The top window is for creating a new alarm or message. It has four input fields: 'Type' (set to 'Alarme'), 'Libellé' (set to 'fenetre ouverte'), 'Source événement' (set to 'salle 445'), and 'Code de gravité' (set to '1'). A 'Send' button is at the bottom left. On the right is a vertical panel with a list: '0 > Information' and '1 > Alarme', with '1 > Alarme' selected. The bottom window displays the details of the selected alarm. It shows 'Alarme' and the date 'jeudi 13 janvier 2005 16:41:8'. Below this, it shows 'fenetre ouverte' and 'salle 445'. There are two buttons: 'Prise en compte' and 'Solder'. Below these buttons, it shows 'Pris en compte par' (rantanplan) and 'Pris en compte le' (jeudi 13 janvier 2005 16:42:10). At the bottom, it says 'Soldé par'.

L'interface proposée n'est qu'un exemple incomplet. Vous veillerez à faire mieux. Les champs inutiles ne sont pas affichés (gravité pour un message ou soldé par pour une alarme non soldée).

De même les boutons ne sont visibles que si nécessaire.

Vous ajouterez des boîtes de messages qui seront affichées quand nécessaires. Vous créerez une classe pour cela, à l'aide de la classe Dialogue en créant une fenêtre modale. Ces boîtes sont affichées quand on valide un message ou une alarme non conformes, ou quand l'opérateur rentre un nom d'opérateur incorrect.

Vous devez blinder l'interface pour que l'opérateur ne puisse pas prendre en défaut votre application.

Troisième partie : Rendre un objet métier écoutable

Nous allons tout d'abord voir pourquoi la solution proposée pour l'exercice des couleurs n'est pas optimale. Nous allons ensuite observer comment se comporte l'atelier de développement java que nous utilisons tous les jours.

Nous allons alors expliquer comment rendre un objet métier écoutable, au même titre qu'une fenêtre écoute un bouton. Ensuite je vous propose les grandes lignes pour traiter avec cette méthode le problème des couleurs. Vous complèterez cette approche, et vous ferez fonctionner le programme correspondant.

Enfin nous allons compléter notre projet de traitement de la supervision des alarmes et messages en rendant l'historique écoutable, et en rajoutant des vues (statistiques faites avec des camemberts) à notre application.

1) Critique de la solution de l'exercice des couleurs.

La solution préconisée pour l'exercice des couleurs, bien que construite en respectant les règles de conception objet, à un inconvénient majeur : chaque panel comporte bien le savoir faire qui lui est propre, sauf le traitement lié à l'événement sur le contrôle qui est dans la fenêtre principale. Le savoir faire du panel est donc coupé en deux.

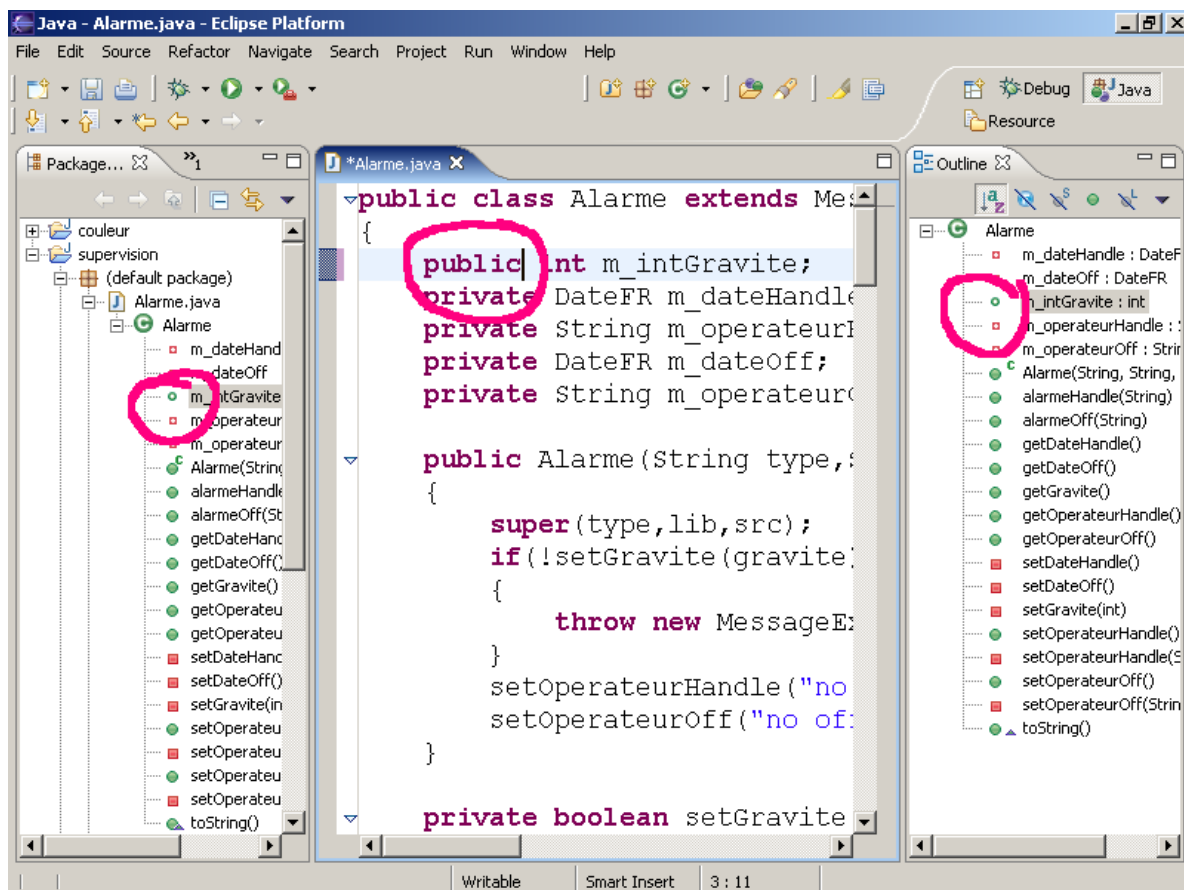
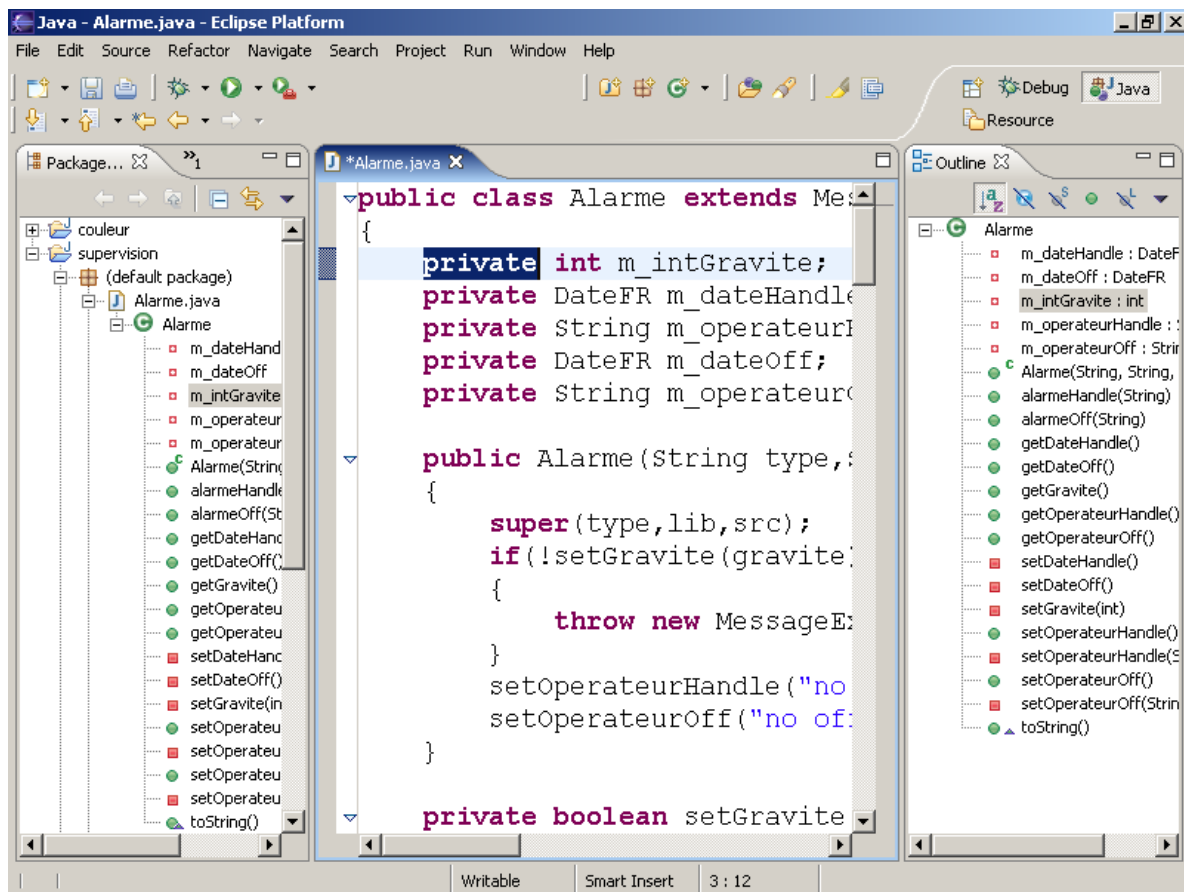
L'avantage de regrouper le traitement lié à l'activation du contrôle avec le panel qui le contient est double : cela permet de diminuer le couplage, et d'augmenter la cohérence. C'est à ce prix que l'on pourrait faire de la programmation de composants « plug and play ». Imaginez, au cours de l'exécution du programme, la possibilité de rajouter différents composants (dans leurs panels respectifs) à la fenêtre principale (typiquement grâce à des cases à cocher dans un menu). Quand un nouveau panel est rajouté, il est à l'écoute du changement de couleur, d'où qu'il vienne, et son activation change la couleur de l'ensemble des autres composants. De la science fiction me dites vous ? Regardons ce qui se passe dans un atelier de développement de type Eclipse par exemple.

2) Comportement de l'interface de l'atelier Eclipse.

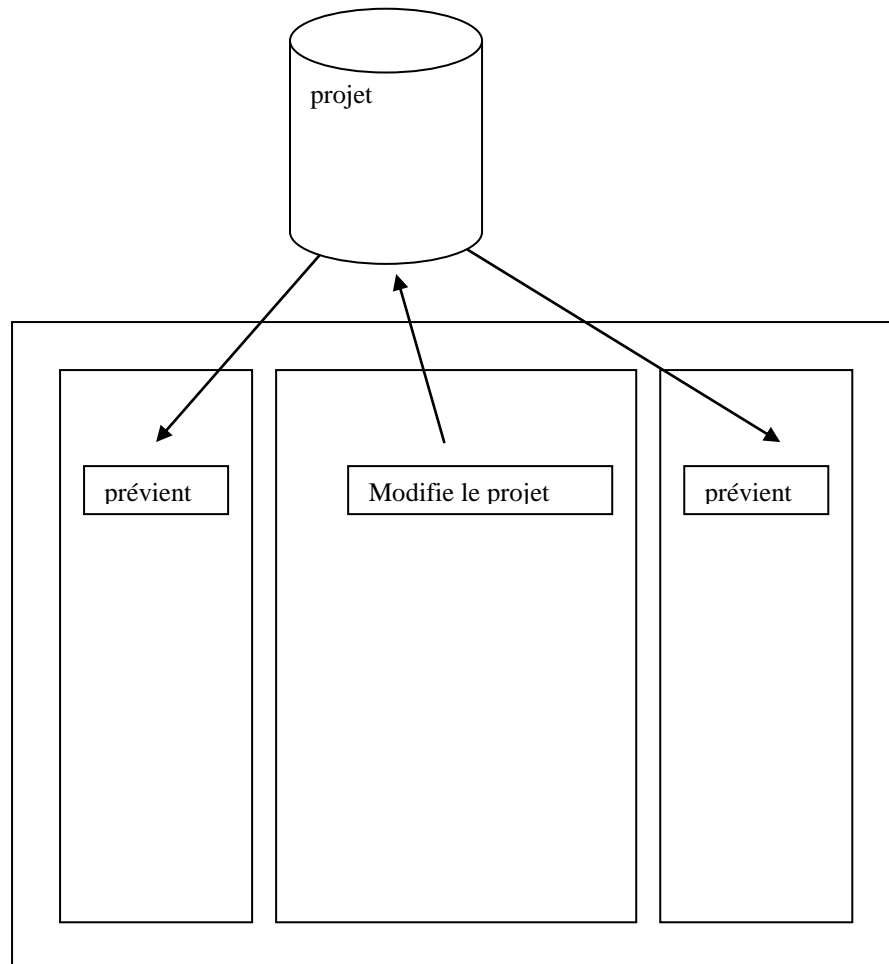
Quand nous travaillons sur un projet java avec Eclipse, notre document de travail est un projet, sur lequel nous avons plusieurs vues.

Ces différentes vues sont synchronisées, si nous changeons un attribut d'une donnée membre, les différentes vues du projet sont automatiquement mises à jour, si elles sont présentes à l'écran. Ces vues sont « plug and play ». Les vues sur un projet sont nombreuses, et nous ne pouvons pas imaginer que la fenêtre principale écoute chaque vue, pour remettre à jour chacune des vues présente à l'écran qui est impactée par le changement. Ce mécanisme serait trop lourd (sachant que le programmeur peut dynamiquement fermer n'importe laquelle de ces vues). La fenêtre devrait aussi s'occuper de la mise à jour des vues, ce qui est contraire aux principes de l'objet (en gros chacun s'occupe de ses affaires et les objets seront bien gardés).

Dans l'exemple suivant, l'attribut **m_intGravite** passe de **private** à **public**. Les vues associées ont été mises à jour (vous le vérifierez sur votre atelier).



Chaque vue va se mettre à jour par rapport au projet. Cela suppose que chaque vue soit prévenue que le projet a changé (et si possible quel type de modification a été fait). Ainsi chaque vue va écouter le projet, et chaque vue aura son écouteur. Cela signifie que le projet, qui est un objet métier a implémenté un mécanisme d'écoute. Ce mécanisme doit permettre de s'abonner à l'écoute du projet, pour être averti des différentes modifications apportées au projet qui intéresse chaque vue.



- 3) Nous allons voir ce qu'il faut mettre en place pour qu'un objet métier puisse être écoutable.

Nous allons construire un tableau qui établit de manière informelle le parallèle entre l'écoute d'un bouton, et l'écoute de mon projet, ceci pour nous faire intuitivement ce que nous devons ajouter à un objet métier pour le faire devenir écoutable. Ensuite nous prendrons l'exemple des couleurs pour réaliser l'écoute sur un objet métier.

Ecoute d'un Button	Ecoute de l'objet métier projet
Interface ActionListener - public void actionPerformed	Interface ProjetListener - public void attributAdded - public void attributModified - public void méthodeAdded - public void méthodeModified - ...
Une classe qui décrit l'événement Class ActionEvent	Une classe qui décrit l'événement Class ProjetEvent
La classe Button possède - méthode addActionListener - méthode removeActionListener - une liste d'écouteurs	La classe Projet doit posséder - méthode addProjetListener - méthode removeProjetListener - une liste d'écouteurs
La classe Button possède un moyen de prévenir les écouteurs que le bouton a été cliqué. - méthode fireActionPerformed	La classe Projet doit posséder un moyen de prévenir les écouteurs qu'un attribut a été ajouté, ou modifié, ou qu'une méthode a été ajoutée, ou modifiée, ... - méthode fireAttributAdded - méthode fireAttributModified - méthode fireMéthodeAdded - méthode fireMéthodeModified - ...

4) Ce tableau nous dévoile le profil que doit avoir un objet métier écoutable. Je vous propose de prendre l'exemple des couleurs et de mettre en place sur cet exemple un objet métier écoutable.

Quel va être l'objet métier que tous les panels veulent écouter pour se mettre à jour quand il a changé ? C'est la couleur bien sûr. Nous allons créer un objet métier couleur (sauf pour ceux qui avaient déjà créé leur HashTable de couleurs, qui travaillerons dans cette classe), et le rendre écoutable par tous, panels et fenêtre principale. Puis nous allons traiter les panels et la fenêtre pour que chacun écoute et modifie sa couleur quand la couleur change, mais aussi que chacun modifie la couleur quand son contrôle lui demande de le faire (ce qui aura pour effet de prévenir chacun que la couleur a changé, donc de repeindre tous les panels et la fenêtre principale).

Tout d'abord nous allons créer l'objet métier Couleur qui représente la couleur de notre fenêtre. Il est à noter que nous avons rajouté le pattern singleton, qui est le moyen d'avoir l'accès à une instance unique d'objet qui est la même pour l'ensemble de l'application (la couleur est unique et la même pour tout le monde, donc plutôt que de faire un accès complexe à cette entité pour garantir l'unicité de cet objet, on applique le pattern singleton).

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

```

public class Couleur
// classe permettant de rendre l'objet métier couleur de la fenêtre
{
    private static Couleur color;        // singleton sur l'objet couleur
    private Color colorie;               // couleur des vues
    private Vector<CouleurListener> v = new Vector<CouleurListener> (5,2);
                                        // liste des abonnés

    private Couleur (Color c) // ici le constructeur est privé car le getInstance est utilisé
    {
        setColorie(c);
    }

    public static Couleur getInstance()
    // permet d'avoir l'instance unique
    {
        if(color == null)
        {
            color = new Couleur(Color.red);
        }
        return color;
    }

    public void addCouleurListener(CouleurListener cl)
    // abonnement pour être prévenu du changement de couleur
    {
        if ( cl != null) v.addElement(cl);
    }

    public void removeCouleurListener(CouleurListener cl)
    // désabonnement
    {
        v.removeElement(cl);
    }

    private void fireCouleurChanged()
    // prévenir tous les abonnés
    {
        // construire une instance d'un objet CouleurEvent
        CouleurEvent ce = new CouleurEvent(this);
        // prévenir tous les abonnés que la couleur a changé
        for ( CouleurListener cl : v)
        {
            cl.couleurChanged(ce);
        }
    }
}

```



```

public Color getColorie()
// permet de connaître la couleur a appliquer
{
    return colorie;
}
public void setColorie(Color c)
// permet de modifier la couleur
{
    colorie = c;
    // il faut donc prévenir tous les intéressés que la couleur a changé
    fireCouleurChanged();
}
}

```

Nous allons maintenant mettre en place l'interface permettant de créer des écouteurs de couleurs.

```

import java.util.*;

public interface CouleurListener extends EventListener
// interface des écouteurs de couleurs
{
    public void couleurChanged(CouleurEvent ce);
}

```

Nous allons définir la classe des événements de changement de couleur.

```

import java.util.EventObject;

public class CouleurEvent extends EventObject
// classe décrivant l'événement de changement de couleur
{
    public CouleurEvent(Couleur source)
    {
        super(source); // l'objet à l'origine de l'événement doit être passé au
                        // constructeur de la classe mère
        // dans ce constructeur nous aurions pu ajouter d'autres informations,
        // et les méthodes get associées à ces informations pour quelles soient
        // récupérées et traitées par l'écouteur ( pensez aux coordonnées de la
        // souris pour les MouseEvent ).
    }
}

```

La fenêtre va écouter l'objet couleur, pour se mettre à jour quand la couleur change.

```

import java.awt.*;

```

```

public class FenColor extends Frame
{
    private PanButton panBut;
    private PanRadio panRadio;
    private PanRight panRight;

    public FenColor()
    {
        // création d'un écouteur de changement de couleur
        EcChangeColor ecChCol = new EcChangeColor();
        // récupération de l'instance unique de couleur
        Couleur c = Couleur.getInstance();
        // notre fenêtre écoute la couleur pour réagir quand la couleur change
        c.addCouleurListener(ecChCol);
        // écouteur de fermeture de fenêtre
        EcWindowClose ecCl = new EcWindowClose();
        addWindowListener(ecCl);

        setLayout(new GridLayout(1,3,40,20));
        panBut = new PanButton();
        panRadio = new PanRadio();
        panRight = new PanRight();

        add(panBut);
        add(panRadio);
        add(panRight);

        setSize(500,400);
        setVisible(true);
        // initialisation de la couleur
        c.setColorie(Color.blue);
    }

    public Insets getInsets()
    {
        return new Insets(40,20,20,20);
    }

    class EcChangeColor implements CouleurListener
    // la classe écouteur implémente la nouvelle interface définie
    {
        public void couleurChanged(CouleurEvent ce)
        // la méthode est appelée chaque fois que la couleur a changé
        {
            setBackground(((Couleur)ce.getSource()).getColorie());
        }
    }
}

```

Le panel des boutons va également se mettre à jour quand la couleur change, mais quand un bouton est pressé, il change la couleur, alors tout le monde est prévenu, et tout le monde se met à jour.

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

public class PanButton extends Panel
{
    private Button butRed, butBlue, butGreen, butYellow ;

    public PanButton()
    {
        // Ecouteur de changement de couleur
        EcChangeColor ecChCol = new EcChangeColor();
        // récupération de l'instance unique de couleur
        Couleur c = Couleur.getInstance();
        c.addCouleurListener(ecChCol);
        // ecouteur sur les boutons
        EcBouton ecButton = new EcBouton();

        setLayout(new GridLayout(4,1,60,20));

        butRed = new Button("Rouge");
        butBlue = new Button("Bleu");
        butGreen = new Button("Vert");
        butYellow = new Button("Jaune");

        butRed.addActionListener(ecButton);
        butBlue.addActionListener(ecButton);
        butGreen.addActionListener(ecButton);
        butYellow.addActionListener(ecButton);

        add(butRed);
        add(butBlue);
        add(butGreen);
        add(butYellow);
    }

    public void setBackground(Color c)
    {
        String name = ColorTools.colorToString(c);
        super.setBackground(c);
        setFocusButton(name);
    }

    private void setFocusButton(String name)
    {
        if(name.equals("Rouge"))butRed.requestFocus();
    }
}
```

```

        else if(name.equals("Bleu"))butBlue.requestFocus();
        else if(name.equals("Vert"))butGreen.requestFocus();
        else if(name.equals("Jaune"))butYellow.requestFocus();
    }

    // classe d'écouteur de changement de couleur
    class EcChangeColor implements CouleurListener
    {
        // méthode appelée quand la couleur a changé
        public void couleurChanged(CouleurEvent ce)
        {
            setBackground(((Couleur)ce.getSource()).getColorie());
        }
    }

    // classe d'écouteur des boutons
    class EcBouton implements ActionListener
    {
        // méthode appelée quand un bouton est appuyé
        public void actionPerformed(ActionEvent ev)
        {
            Couleur c = Couleur.getInstance();
            c.setColorie(ColorTools.stringToColor((String)ev.getActionCommand ()));
        }
    }
}

```

Exercice : Terminez cette application pour que tous les panels soient opérationnels.

Vous noterez que dans chaque Panel il y a des instructions qui se répètent : chaque Panel écoute la couleur. Il faut donc créer une classe PanelCouleur abstraite qui écoute la couleur. Tous les panels hériteront de cette classe, ce qui leur enlève le souci d'écouter la couleur.

Vous pouvez également faire mieux : une Machine à peindre (classe qui peint le Container auquel elle est associée quand la couleur change). Maintenant notre PanelCouleur comporte une machine à peindre, mais notre fenêtre principale aussi...

5) Supervision des messages et alarmes

Nous allons compléter notre application de gestion des messages et alarmes pour pouvoir tenir des statistiques sur la proportion de messages et alarmes, ainsi que sur les états des alarmes (nouvelles, prises en compte, soldées).

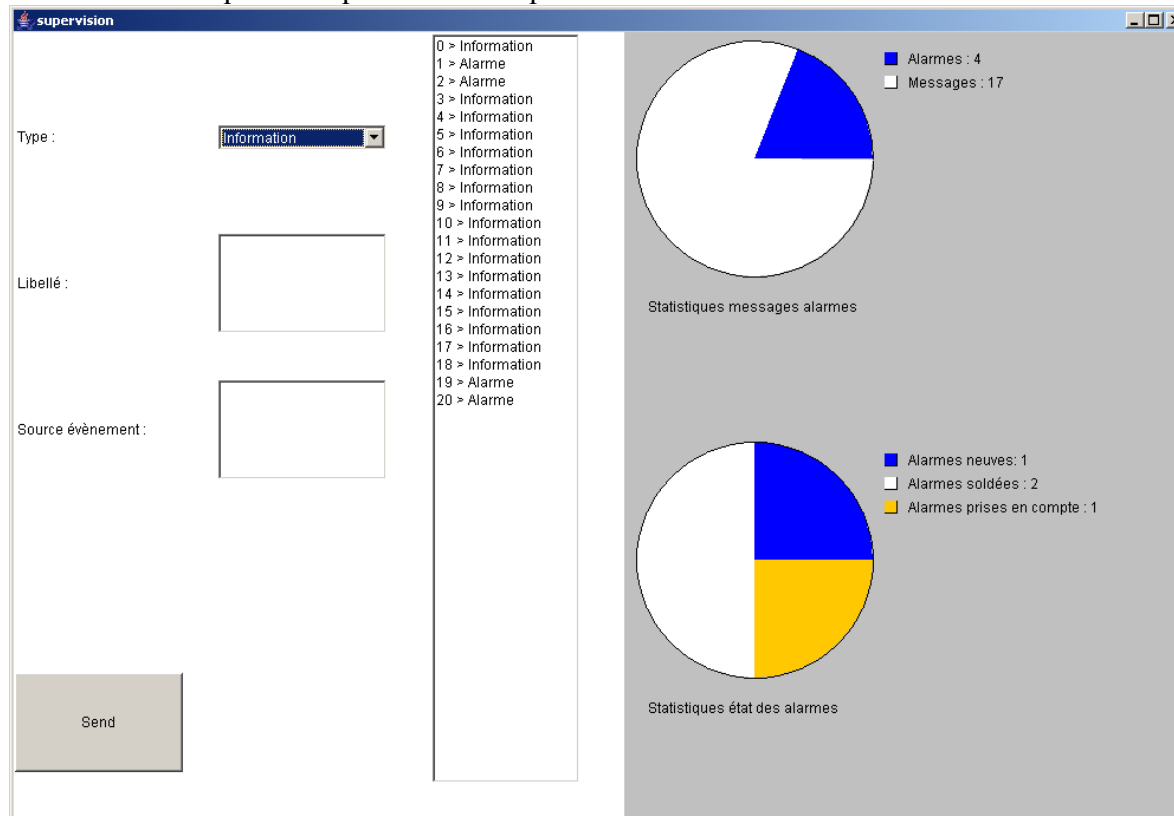
Cela rajoute deux vues à notre application. Ces vues doivent se mettre à jour automatiquement, quand l'historique enregistre une nouvelle information, ou quand l'historique a l'état de l'une de ses alarmes qui est modifié.

Nous allons donc rendre l'objet métier historique écoutable, de même nous allons lui associer le pattern singleton. Puis nous modifierons l'application pour que la mise à jour de la liste soit automatique quand un message ou une alarme a été rajouté à l'historique.

Nous allons ensuite créer un objet Camembert, qui permet de construire un graphe de statistiques. Cet objet devra être bien paramétré pour que nos camemberts intègrent une légende, un titre, et que l'on puisse choisir les couleurs associées.

Nous allons maintenant rajouter deux vues, contenant chacune un camembert, qui seront mises à jour automatiquement quand l'historique subit des variations.

Voici une interface possible pour notre supervision, les mises à jour des différentes vues se faisant automatiquement quand l'historique évolue.



Il n'est pas question de tout développer à la fois : il va falloir être très prudent, et développer de manière très progressive cette application :

- 1) Nous allons mettre en place le pattern singleton pour l'historique. Ainsi le panel de simulation de déclenchement d'un message ou d'une alarme écouterait elle-même son bouton de création de message, et l'historique n'est plus passé en paramètre à qui que ce soit. Par contre la liste n'est plus mise à jour quand un message est créé.
- 2) Nous allons rendre l'historique écoutable, afin de pouvoir mettre à jour la liste des messages et alarmes.
- 3) Les alarmes vont être écoutables, ainsi quand une alarme est prise en compte ou soldée par l'interface, elle prévient qui veut le savoir qu'elle a changé d'état. Plusieurs choix s'offrent à nous, l'historique écoute les alarmes, et l'historique prévient tout ceux qui sont intéressés qu'une alarme a changé d'état (solution simple mais centralisatrice), ou directement ceux qui sont intéressés écoutent les alarmes. Nous noterons au passage que cela permet de prendre en compte ou solder une alarme par un dispositif déporté par rapport à la supervision, qui devrait alors se mettre à jour automatiquement. Je conseille de faire une écoute statique sur les alarmes, pour avoir un seul abonnement pour toutes les alarmes (vecteur d'abonnés et

méthodes d'abonnement et désabonnement statiques), mais c'est bien l'alarme concernée qui nous prévient (méthode fire non statique).

4) quand tout ce fonctionnement a été validé, mettre dans la supervision deux panels qui affichent les compteurs de messages et alarmes, ainsi que le nombre d'alarmes dans chaque état. Validez le fonctionnement de l'application.

5) Vous réaliserez un camembert. Ce camembert devant pouvoir servir dans d'autres applications, nous allons étudier à part la réalisation de ce composant camembert. Ce camembert est réalisé dans un projet distinct, testé sur un ensemble de valeurs qui permet de valider son fonctionnement, puis livré par un jar (voir comment réaliser un jar, et comment utiliser un jar construit ailleurs p 52).

6) Nous allons greffer notre camembert dans l'application du 4. Cette opération doit indolore, et la greffe doit prendre immédiatement si le travail a été fait correctement à l'étape 5. Si vous n'avez pas le temps de créer un objet camembert, vous pouvez greffer celui d'un collègue. Cela validera son travail, mais également votre capacité à insérer un composant dans votre application.

Détail de la création d'un camembert :

1) Notion de pattern MVC

Quand vous créez un composant graphique, ce composant doit être décomposé en trois parties :

- le Modèle des données qui représente les informations qui commandent l'affichage du composant. C'est la classe maîtresse du composant car l'utilisateur modifie l'affichage du composant en modifiant les données du modèle (mise à jour automatique du graphique en fonction des données du modèle).
- La Visualisation qui représente la partie graphique du composant. Cette visualisation est souvent créée en lui donnant un modèle de données. La visualisation écoute les données et répercute tout changement des données sur le visuel.
- Le Contrôle qui gère les opérations que l'opérateur peut effectuer sur le composant. Le contrôle écoute les événements qui arrivent sur le composant pour mettre à jour le composant (visuel et / ou données).

Prenons l'exemple d'un Slider très simplifié (un composant pour régler un volume par exemple). Son Modèle de données contient les min et max et la valeur courante. La vue affiche un curseur correspondant à la valeur courante relativement aux min et max. Le contrôleur récupère les événements souris sur le curseur, pour modifier le modèle de données (et automatiquement la vue se mettra à jour).

Nous allons appliquer ce modèle à notre composant camembert.

2) Application du pattern MVC à notre camembert

Tout d'abord regardons la nécessité de développer notre camembert dans une application distincte. Notre volonté est de créer un composant utilisable dans d'autres applications. Il n'est donc pas question d'avoir une quelconque référence à l'historique ou d'écouter les alarmes dans notre camembert. La meilleure manière d'en être sûr est de développer le camembert dans une autre application, et d'écrire un petit programme de test de notre camembert (par contre il faut vraiment le tester !!!).

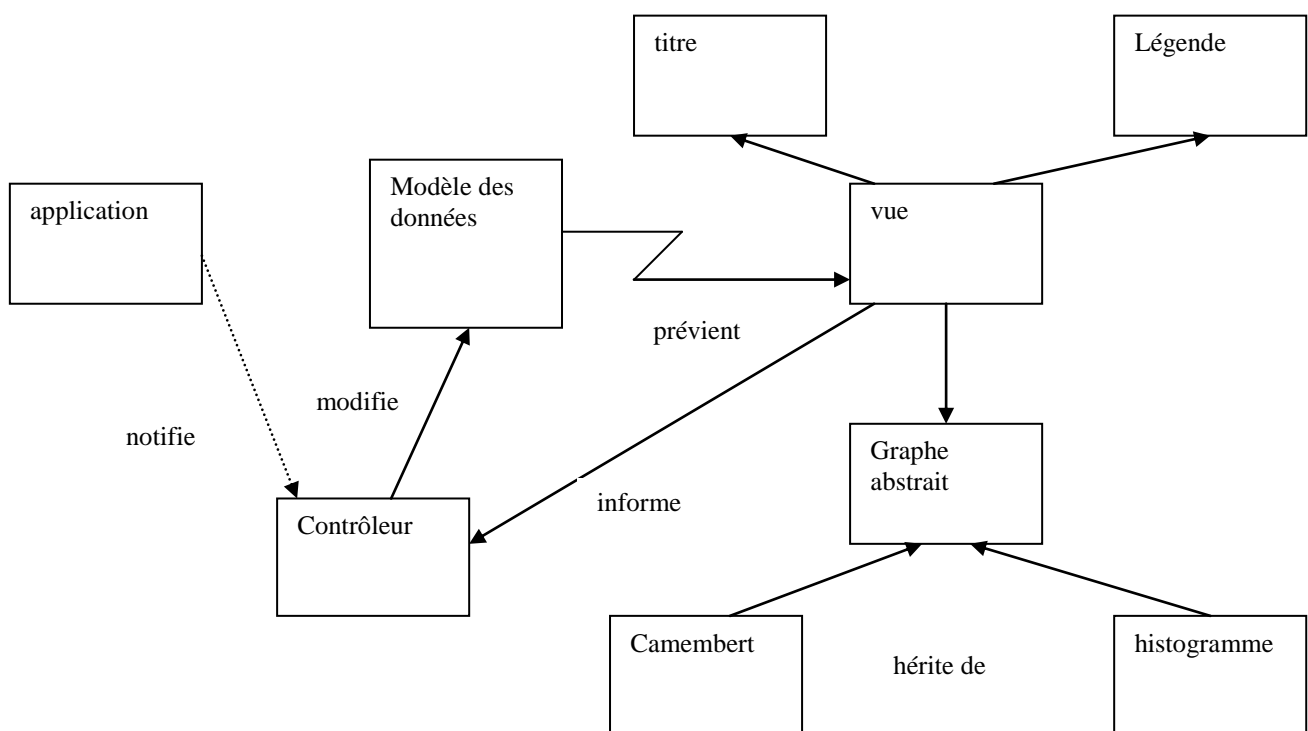
Notre camembert est un composant passif, il n'aura donc pas de contrôleur. Les données comprendront un titre (titre du diagramme) et une collection de portions de camembert,

chaque portion étant composée d'une valeur, d'une couleur, et d'une légende. Eventuellement nous aurons aussi le type du graphique, si nous intuitions déjà qu'un camembert et un histogramme c'est quasiment la même chose...

Il va par contre falloir définir précisément les opérations que l'on veut effectuer sur notre camembert. Changement de type de graphique, changement d'une portion de camembert (de toute la portion, ou seulement de la valeur), ajout d'une portion, modification de plusieurs portions en une seule opération... Faites des choix simples et rationnels.

Je conseille de prendre un Panel comme objet de base de la vue. Ce panel va lui-même contenir trois panels. Un pour le graphe, un pour la légende, et un pour le titre.

Voici un petit schéma des classes en jeu :



Il vous faudra d'abord valider l'initialisation du camembert, puis les évolutions des valeurs. Faites attention aux erreurs d'arrondis, qui provoquent souvent l'affichage d'une minuscule portion de camembert blanche...

Faites bien vivre votre camembert pour être sûr qu'il fonctionne.

Puis ajoutez la possibilité de commuter entre l'histogramme et le camembert, tout au long de votre test.

Quand tout est validé, préparez la livraison de votre camembert, pour l'intégration à notre application.

Quatrième partie : Accès aux bases de données avec JDBC

De manière générale, il vous est demandé, en complément des informations apportées, de rechercher dans l'aide un maximum d'informations, et de parcourir les classes citées pour repérer les informations intéressantes.

Les exercices seront faits à partir de la base de données « banque » et pourront être les mêmes qu'en SQL, la partie requête n'étant pas ici le point crucial de l'apprentissage. Les principales tables de la base de données sont jointes afin de pouvoir comprendre les exemples. Il est important de noter que les outils de développement sont les outils de Sun. Pour utiliser les outils de Microsoft, il faudrait prendre le driver adéquat.

1) Qu'est-ce que JDBC.

1) *Introduction.*

JDBC est l'API java pour se connecter à des bases de données hétérogènes. JDBC est conçue pour que le développeur puisse se concentrer au maximum sur son application, et perde le moins d'énergie possible pour traiter des problèmes techniques de liens avec la base de données.

JDBC à trois rôles :

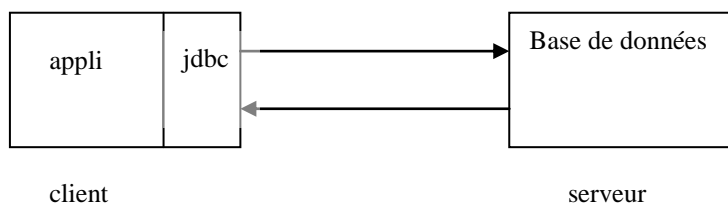
- Se connecter à une base de données avec l'interface adéquate (driver).
- Envoyer des requêtes SQL.
- Exploiter les résultats des requêtes.

ODBC ne pourrait pas être employé dans java car son interface est en langage C, ODBC n'est pas objet, et son emploi n'est pas très simple. De plus ODBC nécessite d'installer un driver ODBC sur la machine cliente.

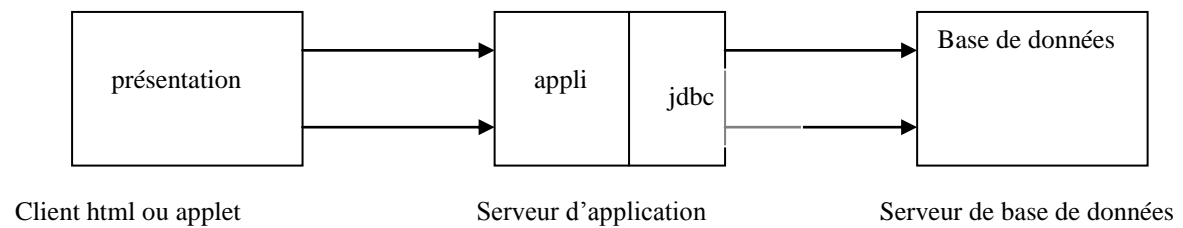
Avec JDBC nous avons une solution objet, simple, en pur Java, qui ne nécessite rien de particulier sur la machine cliente (à condition de prendre le driver JDBC qui n'est pas encore adopté par Bill).

2) *Technologies.*

Technologie 2/3 ou client-serveur.



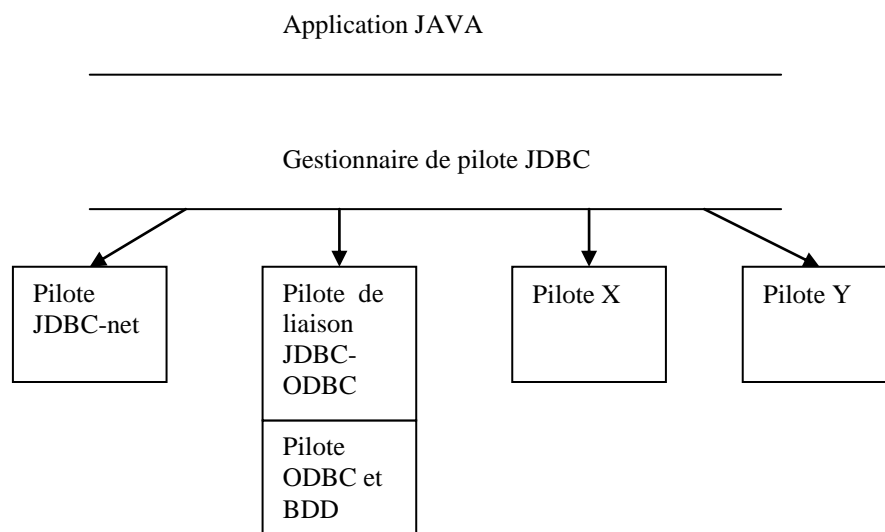
Technologie 3/3 ou serveur d'application et serveur de base de données



L'avantage de la technologie 3/3 est de permettre de sérier les problèmes. Cela permet encore de décharger le client et de le banaliser (explorateur internet), et d'améliorer les performances globales du système (client plus réduit, moins d'informations sur les lignes, ...).

JDBC est conforme au standard SQL.

Voici un schéma du fonctionnement de java :



Le pilote JDBC ne nécessite pas d'installation sur la machine cliente, et il est de plus en plus intégré aux bases de données.

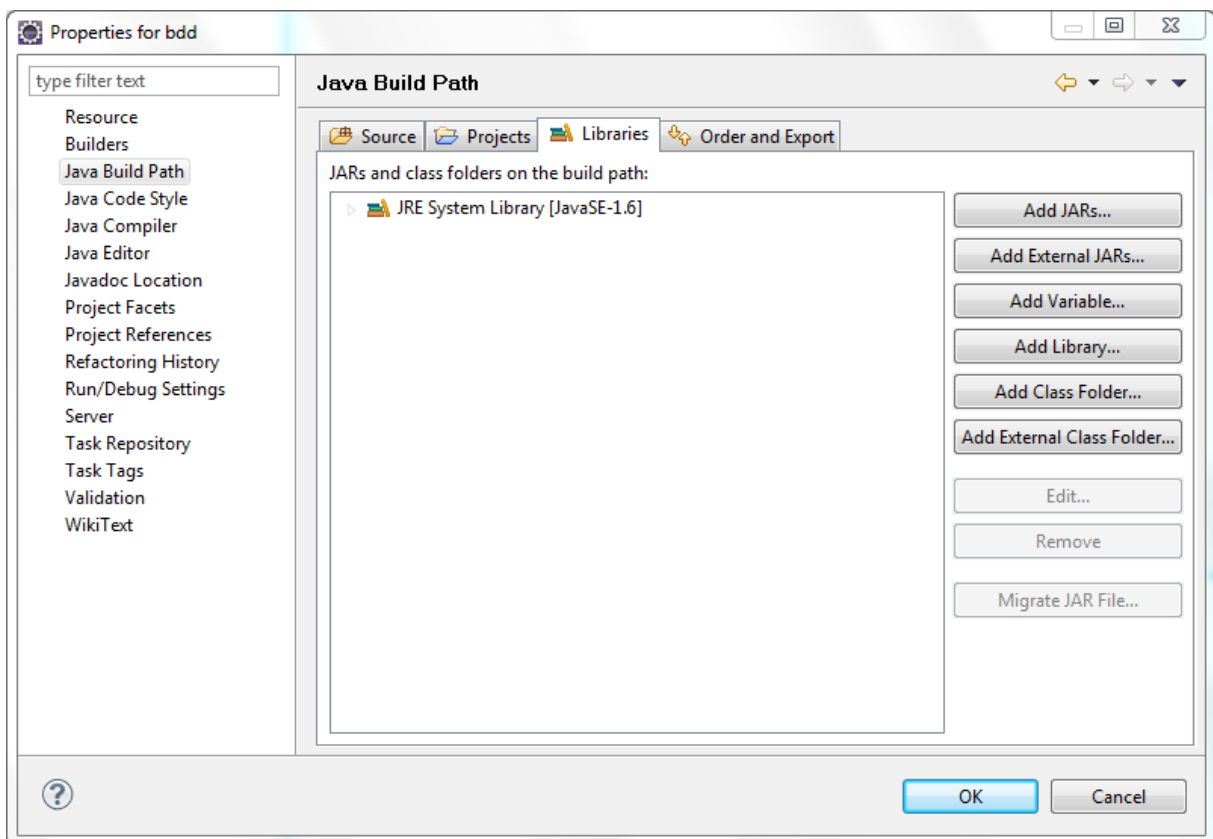
Le pilote ODBC lui doit être installé sur les machines clientes.

2) La connexion aux bases de données.

La classe DriverManager est le gestionnaire des drivers permettant à l'utilisateur l'accès aux bases de données. Les bases de données intégrant un driver JDBC directement, nous pouvons directement dialoguer avec les bases de données via ce driver.

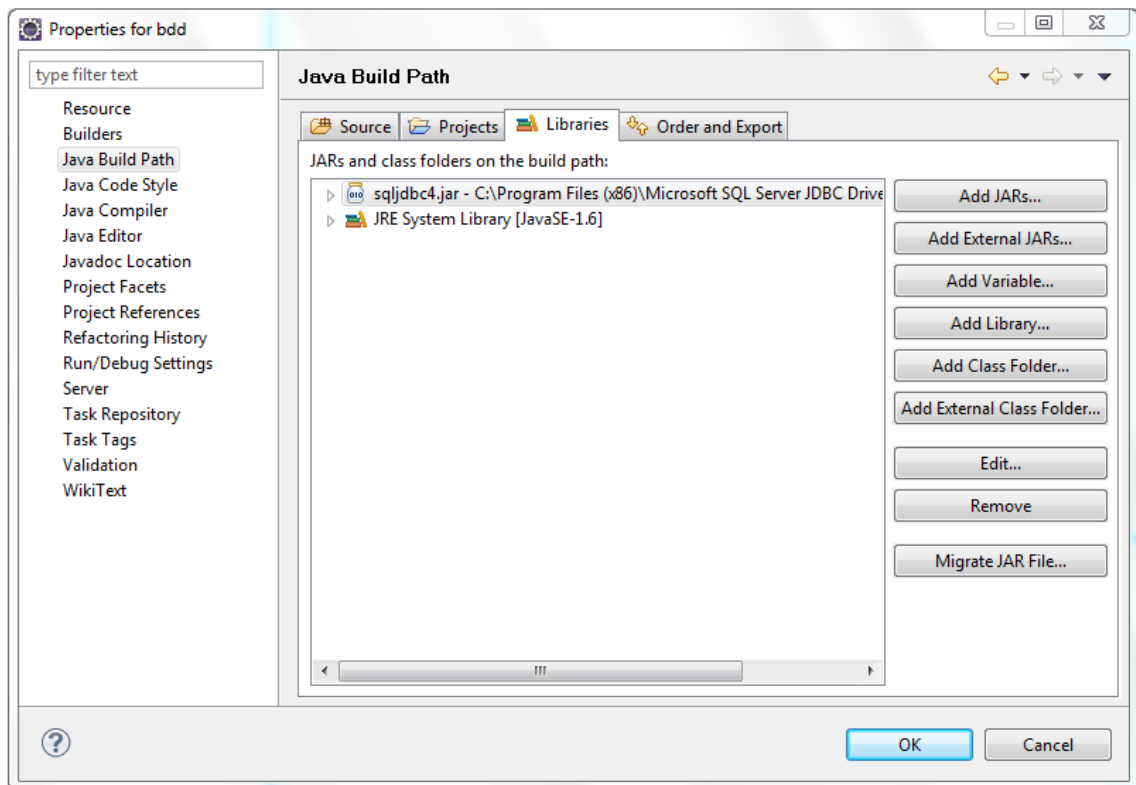
1) Installation du driver JDBC sur la plateforme de développement.

Il faut installer le driver JDBC en ajoutant le JAR sqljdbc4.jar au projet. Positionnez vous dans project / properties/ java build path onglet Libraries, vous obtenez la fenêtre suivante :



Il vous suffit de cliquer sur Add External JARs... pour ajouter le fichier sqljdbc4.jar que vous trouvez dans documents/objet/BDD.

Maintenant votre environnement de développement dispose du driver jdbc.



2) *Chargement de la classe du Driver désiré dans le programme.*

```
try
{
    Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
}
catch ( Exception e)
{
    ...
}
```

La classe statique désirée est chargée. A l'établissement de la connexion le driver sera automatiquement chargé par la fonction getConnection.

3) *Etablissement de la connexion*

Pour se connecter à la base de données il suffit de demander une connexion à la base au DriverManager en donnant le type de service demandé (driver jdbc sqlserver), le nom du serveur (serveur- msoft) le port de connexion (1433) le nom de la base de données (AirRogerio) le nom d'utilisateur (visiteur) et son mot de passe (visiteur)..

```
String url = "jdbc:sqlserver://serveur-msoft:1433; databaseName= AirRogerio;";
Connection cox = null;
```

```

try{
    cox == java.sql.DriverManager.getConnection(url,"visiteur", "visiteur");
        // demande d'une connexion à la base avec le nom d'user et son mot de passe
    }
catch(Exception e)
{
    ...
}

```

Exemple de connexion à une base de données:

```

// connexion à une base de donnée
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Essai
{
    public static void main(String [] arg)
    {
        try
        {
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

            String url = "jdbc:sqlserver://serveur-msoft:1433;" +
                "databaseName=AirRogerio;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"visiteur", "visiteur");
            Statement st= con.createStatement();
            ResultSet rs = st.executeQuery("select * from pilote");
            while (rs.next())
            {
                String s = rs.getString(1);
                System.out.println(s);
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

Une autre solution pour créer la connexion est celle-ci :

```
try
{
    Class dri=Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
    // classe décrivant le driver de connexion

    Driver dDriver=(java.sql.Driver)dri.newInstance();
    // obtention d'une instance de cette classe

    DriverManager.registerDriver(dDriver);
    // enregistrement de cette instance dans le gestionnaire

    String u="jdbc:sqlserver://serveur-msoft:1433;databaseName=AirRogerio;";
    Connection con = DriverManager.getConnection(u,"visiteur", "visiteur");
    Statement st= con.createStatement();

    System.out.println ("ça marche ");
}
catch(Exception e)
{
    System.out.println ("la connexion a échoué");
    e.printStackTrace ();
}
```

3) Passage d'une requête et exploitation des résultats

Ici nous allons passer une requête à la base de données et exploiter les éventuels résultats. Dès que nous avons récupéré une connexion, nous allons pouvoir passer des requêtes à la base. Nous allons pouvoir travailler de trois manières différentes.

- Les requêtes directes à la base (Statement)
- Les requêtes préparées (PreparedStatement). Ce sont des requêtes, souvent paramétrées, qui sont précompilées par la base, ce qui accélère leur traitement. Nous utiliserons des requêtes préparées chaque fois qu'une requête doit être passée sur un grand ensemble de données.
- L'appel de procédures stockées. Ces procédures peuvent être paramétrées, avoir un résultat en retour, et nous retourner un ensemble solution (les lignes résultant d'un SELECT par exemple).

Nous verrons également comment exploiter les résultats des requêtes.

1) Requetes directes.

Voici un exemple de requête, une fois que nous avons eu une connexion (cox).

```
Statement st = cox.createStatement(); // creation d'un objet requête directe
ResultSet resultat; // création d'une variable qui référencera l'ensemble des résultats
```

```
resultat = st.executeQuery("SELECT x,y,z FROM table");
// exécution de la requête
// nous exploiterons plus loin les résultats
int ret = st.executeUpdate("DELETE ...");
```

Il existe trois manières d'exécuter des requêtes SQL.

- ExecuteQuery : c'est une interrogation qui produit un ensemble simple de lignes résultat (SELECT).
- ExecuteUpdate : c'est la modification de l'état de la base (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE). Le résultat est la modification de 1 ou plusieurs colonnes de 0 à plusieurs lignes de la table. Cela retourne le nombre de lignes modifiées (pour les requêtes CREATE et DROP le résultat est toujours 0).
- Execute : est utilisé pour exécuter des requêtes qui retournent plus d'un ensemble de résultat, ou plus d'une valeur de retour, ou les deux. Cette manière d'exécuter des requêtes SQL est assez peu utilisée par les développeurs. Nous y jetterons toutefois un œil, il arrive qu'une procédure stockée génère plus d'un ResultSet et il est intéressant de savoir exploiter chacun de ces ResultSet.

2) Exploitation des résultats.

Un ResultSet contient en retour d'un executeQuery toutes les lignes qui satisfont les conditions. Reprenons notre code:

```
resultat = st.executeQuery("SELECT x,y,z FROM table");
// exécution de la requête
while (resultat.next())
{
```

```

        int i = resultat.getInt("x");
        String S = resultat.getString("y");
        Float f = resultat.getFloat("z");
        System.out.println("ligne = "+i+" "+S+" "+f);
    }

    st.close(); // il est recommandé de fermer la Statement même si le garbage collector fait
                //le travail quand même.
    cox.close(); // idem pour la connexion

```

Le ResultSet à un curseur sur les lignes résultantes. Initialement ce curseur est positionné avant la première ligne. La méthode next() le positionne sur la ligne suivante.

Pour lire la ligne nous utilisons les fonctions:

- getXXX("nom-de-colonne"); ou XXX est le type de la donnée lue (voir la classe ResultSet) .
- getXXX(numcolonne); ou XXX est le type de la donnée lue (voir la classe ResultSet) et numcolonne est le numéro de la colonne concernée (1 pour la colonne la plus à gauche).

Les noms de colonne ne peuvent être utilisés que si ils figurent dans la requête SQL. Si deux colonnes ont le même nom, l'accès par le nom de colonne ne donne que la première des deux. Il est plus efficace de passer par le numéro de colonne, mais il est plus lisible de passer par le nom des colonnes (sauf dans le cas ou le nom est dupliqué).

Ci joint une copie des correspondances entre les types Java et les types JDBC, et un exemple de traitement d'une requête SQL en direct sans résultat, et d'une requête SQL avec un ResultSet.

```

// exemple de requête SQL sans résultat

import java.sql.*;

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            //établissement de la connection à la base 'banque'
            //,où l'identificateur de connection
            // est "arrault" , et de code ""

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");

            //création d'un objet de la classe Statement qui permet
            //d'effectuer des requêtes liées à la connection 'con'
            Statement select=con.createStatement ();

            //appel de la méthode executeUpdate de la classe
            //Statement qui permet d'écrire dans une base
            select.executeUpdate("INSERT INTOCOMPTE (NumCompte,
                Nom, Prenom, Solde) VALUES(13, 'Grisolano',
                'Philippe', 18000)");

            //il est recommandé de fermer l'objet Statement
            select.close ();
            //et de fermer la connection
            con.close ();

        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}

```


// exemple de requête SQL avec un résultat

```
import java .sql .*;

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            Statement select=con.createStatement ();

            //executeQuery correspond à la demande d'exécution de la
            // requête. La variable result ( ResultSet ) est
            // est l'ensemble des résultats renvoyés par la requête
            ResultSet result =select.executeQuery ("SELECT * "+
                "FROM COMPTE");

            //next renvoie true lorsqu'il existe une rangée
            //supplémentaire
            while(result.next())
            {
                //conversion du résultat dans le bon type
                int Num=result.getInt (1);
                String Nom=result.getString (2);
                String Prenom=result.getString (3);
                String argent=result.getString (4);
                System.out.println ("Num : "+Num+"      Nom : "+Nom+"
                    Prenom : "+Prenom+ "      argent : "+argent);
            }

            //il est recommandé de fermer l'objet Statement
            select.close ();
            //et de fermer la connection
            con.close ();

        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```

3) Les requêtes pré compilées (ou paramétrées).

Les requêtes pré compilées sont des requêtes " à trous ", que le SGBD compile afin de préparer leur exécution. Cela permet d'accélérer leur traitement.

La classe PreparedStatement représente ce type de requête. Une PreparedStatement contient une requête pré compilée. Elle a au moins un paramètre en entrée. Ce paramètre est représenté dans la requête par un point d'interrogation "?".

Avant l'exécution d'une PreparedStatement il faut appeler la fonction setXXX pour chacun de ces paramètres (afin de remplir tous les trous).

Les PreparedStatements sont des requêtes exécutées un grand nombre de fois, qui sont pré compilées afin d'en optimiser le traitement.

La classe PreparedStatement hérite de Statement, mais il ne faut pas utiliser les méthodes de la classe mère, mais toujours les méthodes de la classe fille.

Exemple d'utilisation:

```
// préparation de la requête
PreparedStatement ps = cox.prepareStatement("UPDATE table SET m=? WHERE x=?");

// garniture des trous avant l'exécution
ps.setString(1,"toto");
ps.setFloat(2,5.0);

// exécution de la requête
ps.executeUpdate();
```

Pour l'appel suivant nous pouvons redéfinir un ou plusieurs des paramètres, les paramètres non modifiés étant conservés. La fonction clearParameters efface tous les paramètres. Le mode de fonctionnement par défaut est le mode autocommit.

setNull() met un paramètre à null.
setxxx () xxx représente le type Java.

Dans le cas d'un SELECT le traitement du ResultSet retourné est le même que pour une requête directe.

Il est à noter que les points d'interrogation de la PreparedStatement ne remplace que des valeurs de champs de la base. Il ne peuvent pas se substituer à des noms de colonne, ou de table; cela serait l'objet de la définition d'une nouvelle requête. N'oublions pas qu'ici nous traitons des requêtes paramétrées (pour plus de détail voir votre documentation SQL préférée).

Voici un exemple de traitement par une requête pré compilée:

// exemple de requêtes paramétrées

```
import java .sql .*; // importation des classes SQL

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            //établissement de la connection à la base 'banque'
            //,où l'identificateur de connection
            // est "arrault" , et de code ""

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");

            //Créer une requête prédéfinie par la classe
            // PreparedStatement sur la connexion con
            PreparedStatement instruction = con.prepareStatement (
                "SELECT * FROM COMPTE WHERE Nom=? ");

            //donner la valeur du paramètre pour l'exécution de la
            //requête
                instruction.setString (1,"perrin" );

            //exécution de la requête (lecture de la base)
            ResultSet result =instruction.executeQuery ();

            //affichage du résultat
            while(result.next())
            {
                String Nom=result.getString (2);
                System.out.println ("Nom : " + Nom);
            }

            // fermeture de la requête
            instruction.close();

            //Créer une requête prédéfinie pour une mise a jour de la
            // base avec cette fois-ci deux paramètres
            instruction = con.prepareStatement (
                "UPDATE COMPTE "+
                "SET Solde=? " +
                "WHERE Nom=? ");

            //donner les valeurs manquant à la requête pour chacun
```

```

//des deux paramètres de la requête paramétrée
instruction.setInt (1,122 );
instruction.setString (2,"perrin" );

//exécuter la requête
instruction.executeUpdate ();

//fermeture de la requête
instruction.close ();

// fermeture de la connexion
con.close ();

}
catch(Exception e)
{
    //ceci permet d'écrire l'exception interceptée
    e.printStackTrace ();
}
}
}

```

4) Procédures stockées.

Une procédure stockée est une procédure enregistrée sur la base de donnée. La classe CallableStatement est la classe qui permet une requête via une procédure stockée. Elle hérite de PreparedStatement. Un objet CallableStatement contient un appel à une procédure stockée.

Nous trouverons deux formes d'appel d'une procédure stockée.

- Les procédures non paramétrées.
- Les procédures paramétrées.

Pour chacun de ces cas il peut y avoir ou pas un code de retour de la procédure stockée

Cela nous donne les quatre interfaces suivantes.

```
{ call nom_proc }  
{ call nom_proc ( ?, ?, ? ) }  
{ ?=call nom_proc }  
{ ?=call nom_proc ( ?, ?, ? ) }
```

Voici comment créer une CallableStatement:

```
CallableStatement cs = cox.prepareCall( "{ call test ( ?, ? ) }");
```

Avant d'exécuter la requête il faut garnir les paramètres.

- Les paramètres en entrée seront affectés en utilisant la méthode setXXX ou XXX est le type JDBC des données. C'est une conversion entre une donnée Java et une donnée JDBC.
- Les paramètres en sortie seront décrits avant l'appel, par l'appel de la méthode registerOutParameter().

setInt(1,7); // le premier paramètre est un entier en entrée. Il vaut 7

```
cs.registerOutParameter(2,java.sql.Types.TINYINT);  
// le paramètre 2 est un Byte en sortie.
```

```
cs.executeUpdate(); // ou ResultSet rs = cs.executeQuery();
```

Il ne reste plus qu'à récupérer la valeur du paramètre en sortie.

```
byte x = cs.getBytes(2);
```

Si la procédure stockée nous retourne un ResultSet, il sera alors traité comme précédemment.

Attention!!!

Si nous traitons le code de retour de la procédure stockée ce sera le premier paramètre de l'appel. Il sera traité comme un paramètre en sortie. Il est important de noter que si nous

avons un ResultSet et une valeur de retour, ou des paramètres de sortie, il faut d'abord vider le ResultSet avant de récupérer les valeurs de sortie, ou de retour.

Vous trouverez des exemples d'appels des procédures stockées dans les 7 cas suivants:

- Pas de paramètres.
- Un paramètre en entrée.
- Un paramètre en sortie.
- Un Paramètre en entrée/sortie.
- Deux ResultSet en résultat de l'exécution de la requête, avec un paramètre en entrée.
- Une valeur de retour de la procédure stockée.
- Une valeur de retour de la procédure stockée avec un ResultSet.

/* exemple 1 : procédure stockée non paramétrée

codage de la procédure sql */

CREATE PROCEDURE phil AS

BEGIN

SELECT *

FROM COMPTE

END

/* exemple 1 : procédure stockée non paramétrée

codage du programme java */

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");

            //Appel d'une procédure stockée(préalablement créée sous
            //sql Server) sans paramètre
            CallableStatement cs= con.prepareCall ("{call phil}");

            //exécution de la procédure stockée
            ResultSet result=cs.executeQuery ();

            // lecture du résultat
            while(result.next())
            {
                int Num=result.getInt (1);
                String Nom=result.getString (2);
                String Prenom=result.getString (3);
                String argent=result.getString(4);
                System.out.println ("Num : "+Num+"  Nom : " +Nom
                    +"  Prenom : "+Prenom+ " argent : "+argent);
            }
            // fermeture de le requête et de la connexion
            cs.close ();
            con.close ();

        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```


/* exemple 2 : procédure stockée avec un paramètre en entrée

codage de la procédure sql */

CREATE PROCEDURE paramin (@num int) AS

BEGIN

SELECT *

FROM COMPTE

WHERE NumCompte = @num

END

/* exemple 2 : procédure stockée avec un paramètre en entrée

codage du programme java */

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            //Appel d'une procédure stockée(préalablement créée sous
            //sql Server) avec paramètre
            CallableStatement cs= con.prepareCall
                ("{call paramin(?)}");

            // passage du premier paramètre
            cs.setInt(1,4);
            //exécution de la procédure stockée(avec un paramètre)
            ResultSet result=cs.executeQuery ();

            //récupération des valeurs renvoyées par la procédure
            while(result.next())
            {

                int Num=result.getInt (1);
                String Nom=result.getString (2);
                String Prenom=result.getString (3);
                String argent=result.getString(4);
                System.out.println ("Num : "+Num+" Nom : "+Nom+
                    " Prenom : "+Prenom+ " argent : "+argent);
            }

            // fermeture de la requête et de la connexion
            cs.close ();
            con.close ();
        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```

/* exemple 3 : procédure stockée avec un paramètre en sortie

codage de la procédure sql */

CREATE PROCEDURE paramout @solde int OUTPUT

AS

BEGIN

SELECT @solde = MAX(Solde)

FROM COMPTE

WHERE Nom = 'Dumousseau'

END

/* exemple 3 : procédure stockée avec un paramètre en sortie

codage du programme java */

```
import java.sql.*; // importation des classes JDBC

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            //Appel d'une procédure stockée avec un paramètre de
            //sortie
            CallableStatement cs= con.prepareCall
                ("{call paramout(?)})");

            //Description du paramètre en sortie
            cs.registerOutParameter (1,java.sql.Types.INTEGER );

            //exécution de la requête
            cs.executeUpdate ();

            //récupération du paramètre
            int Num=cs.getInt (1) ;

            //affichage du résultat
            System.out.println ("Num : "+Num);

            // fermeture de la requête et de la connexion
            cs.close ();
            con.close ();

        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```

/* exemple 4 : procédure stockée avec un paramètre en entrée/sortie et un ResultSet

codage de la procédure sql */

/* ne vous perdez pas dans le sens profond de cette procédure ... elle ne sert qu'à avoir un ResultSet et un paramètre en entrée sortie */

CREATE PROCEDURE paraminout @solde int OUTPUT

AS

BEGIN

SELECT Solde

FROM COMPTE

WHERE NumCompte = @solde

Select @solde = Solde

FROM COMPTE

WHERE Nom = 'Grisolano'

END

/* exemple 4 : procédure stockée avec un paramètre en entrée/sortie et un ResultSet

codage du programme java */

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            //instruction d'appel d'une procédure stockée avec un
            // paramètre d'entrée/ sortie
            CallableStatement cs= con.prepareCall
                ("{call paraminout(?)})");
            //passage du paramètre en entrée
            cs.setFloat (1,12.0f);//si float

            //Description du paramètre en sortie ( le même )
            cs.registerOutParameter (1,java.sql.Types.FLOAT );

            //execution de la requete
            ResultSet rs = cs.executeQuery();
            //récupération des valeurs renvoyées par la procédure
            while(rs.next())
            {
                // vidage du ResultSet
            }

            //récupération du paramètre
            float Solde=cs.getFloat (1) ;

            //affichage du résultat
            System.out.println ("Nouveau solde : "+Solde);

            // fermeture de la requête et de la connexion
            cs.close ();
            con.close ();
        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```

/* exemple 5 : procédure stockée avec un paramètre en entrée et deux résultats

codage de la procédure sql */

CREATE PROCEDURE paramresult @solde varchar(30)

AS

BEGIN

SELECT *

FROM COMPTE

WHERE Nom = @solde

SELECT *

FROM COMPTE

END

**/* exemple 5 : procédure stockée avec un paramètre en entrée et deux résultats
codage du programme java */**

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        try
        {
            //Chargement de la classe du 'Driver'
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            //Appel d'une procédure stockée avec un paramètre en
            //entrée et retournant un ResultSet
            CallableStatement cs= con.prepareCall
                ("{call paramresult(?)})");

            //passage du paramètre en entrée
            cs.setString (1,"perrin");

            //exécution de la requête
            boolean ok=cs.execute ();
            if ( ok )
            {
                do
                {
                    // récupération d'un résultat
                    ResultSet result = cs.getResultSet() ;
                    //affichage d'un champ de la réponse récupérée
                    while (result.next ())
                    {
                        // récupération du troisième champ de la ligne
                        String Num=result.getString (3) ;

                        System.out.println ("Prenoms: "+Num);
                    }
                } while (cs.getMoreResults()) ;
            }
            // fermeture de la requête et de la connexion
            cs.close ();
            con.close ();
        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```


/* exemple 6 : procédure stockée avec des paramètres et un code de retour

codage de la procédure sql */

```
CREATE PROCEDURE param2in1retour @nom varchar(30),@combien int
AS
BEGIN
UPDATE COMPTE set Solde=@combien
WHERE Nom=@nom
RETURN 5
END
```

/* exemple 6 : procédure stockée avec des paramètres et un code de retour

codage de la procédure sql */

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        int code; // valeur du code retour récupéré de l'appel SQL
        try
        {
            //Chargement de la classe du 'Driver'
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con = java.sql.DriverManager.getConnection
                (url,"arrault", "");
            // préparation de l'appel de la procédure stockée
            // ?= call indique que l'on récupère la valeur de retour
            // le premier paramètre in est fourni à l'appel
            // le deuxième paramètre in est fixé pour cette requête
            CallableStatement cs= con.prepareCall
                ("{?=call param2in1retour(?,123)}");

            // configuration du type de la valeur de retour ( c'est
            // forcément un entier )
            cs.registerOutParameter (1,java.sql.Types.INTEGER );

            // la valeur du premier paramètre in est configurée
            cs.setString (2,"perrin");

            // exécution de la requête
            cs.executeUpdate ();

            // récupération du résultat de la valeur de retour
            code=cs.getInt(1) ;

            // fermeture de la requête et de la connexion
            cs.close ();
            con.close ();
            System.out.println ("code retour :"+code);
        }
        catch(Exception e)
        {
            //ceci permet d'écrire l'exception interceptée
            e.printStackTrace ();
        }
    }
}
```

/* exemple 7 : procédure stockée avec un code retour et un ResultSet

codage de la procédure sql */

```
CREATE PROCEDURE paramtotal @nom varchar(30), @max money
OUTPUT, @combien int
AS
BEGIN
UPDATE COMPTE SET Solde=@combien
WHERE Nom=@nom
SELECT *
FROM COMPTE
WHERE Solde=@combien
SELECT @max=MAX(Solde)
FROM COMPTE
RETURN 5
END
```

/* exemple 7 : procédure stockée avec un code retour et un ResultSet

codage de la procédure sql */

```
import java.sql.*; // importation des classes JDBC

public class Essai
{
    public static void main (String[] args)
    {
        int code;
        try
        {
            //Chargement de la classe du 'Driver'
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch (Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {
            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            Connection con =
                java.sql.DriverManager.getConnection(url,"arrault", "");
            // constitution de la requête d'appel à la procédure avec
            // une valeur prédéfinie pour un paramètre (4)
            CallableStatement cs= con.prepareCall(
                "{?=callparamtotal(?,?,123)}");
            //
            // (1) (2) (3) (4)
            // (1) est la valeur retournée par la fonction
            // (2) est le premier paramètre. Il est en entrée
            // (3) est le deuxième paramètre. Il est en sortie
            // (4) est le troisième paramètre. Il est en entrée et
            // fixé dans la requête

            // description de la valeur de retour de la procédure (1)
            cs.registerOutParameter (1,java.sql .Types .INTEGER );

            // description du paramètre en sortie (3)
            cs.registerOutParameter (3,java.sql .Types .DOUBLE );

            // la valeur du premier paramètre in est configurée (2)
            cs.setString (2,"perrin");

            cs.execute ();

            // nous récupérons les retour d'update et les resultats
            ResultSet rs= null;
            int val = 0;
            do
            {
                val = cs.getUpdateCount();
                if ( val != -1) // il y a un update
                {
                    System.out.println("nb valeurs mises à jour:" + val);
                }
            }
        }
    }
}
```

```

rs = cs.getResultSet();

if (rs != null) // il y a un resultat
{
    // parcours du ResultSet jusqu'au bout
    while ( rs.next () )
    {
        // si les résultats sont différents attention
        System.out.println (rs.getInt (1));
    }
}
while ( rs != null && val != -1);
// arrêt quand il n'y a plus de resultat ni d'update

// récupération de la valeur de retour de la procédure
code=cs.getInt(1);

// récupération de la valeur du paramètre en sortie
System.out.println ("le solde max est:"+cs.getDouble(3));

// fermeture de la requête et de la connexion
cs.close ();
con.close ();

System.out.println ("code retour :"+code);
}
catch(Exception e)
{
    //ceci permet d'écrire l'exception interceptée
    e.printStackTrace ();
}
}
}

```

5) Le Contrôle d'intégrité de la base de données.

Il serait temps de s'intéresser à l'intégrité de la base de données. Toutes les requêtes effectuées jusqu'à maintenant ont été effectuées en autocommit. Le mode de fonctionnement est une caractéristique de la connexion. Le mode par défaut est autocommit, c'est à dire que chaque requête SQL est considérée comme une transaction individuelle. Le commit s'effectue quand la requête se termine, ou quand la prochaine exécution démarre (le premier des deux). Dans le cas de requête retournant un ResultSet, le commit s'effectue quand la dernière ligne du ResultSet a été lue, ou que le ResultSet a été fermé.

Les méthodes suivantes sont utilisées (méthodes de la classe Connection):

- `getAutoCommit()` : retourne le mode vrai si autocommit.
- `setAutoCommit(true)` : positionne le mode autocommit à vrai.
- `commit()` : si le mode est non autocommit, cela enregistre les derniers changements depuis le dernier commit/rollback, et relâche tous les verrous posés par la connexion.
- `Rollback()`: si le mode est non autocommit, cela annule les derniers changements depuis le dernier commit/rollback, et relâche tous les verrous posés par la connexion.

Voici un exemple de sécurisation de transaction en utilisant les exceptions, pour ne valider une transaction que si tout c'est bien passé.

// programme java de préservation de l'intégrité de la base de donnée

```
import java.sql.*; // importation des classes jdbc

public class Class1
{
    public static void main (String[] args)
    {
        Connection con=null; // référence de la connexion
        try
        {
            //Chargement de la classe du 'Driver'

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        }
        catch(Exception e)
        {
            System.out.println ("le pilote n'a pu être chargé");
        }
        try
        {

            String url = "jdbc:sqlserver://serveur-sql:1433;" +
                "databaseName=banque;";
            con = java.sql.DriverManager.getConnection (url,"arrault","");
            Statement select=con.createStatement ();

            // spécifions que les transactions devront être validées
            // manuellement
            con.setAutoCommit (false);

            //appel de la méthode executeUpdate de la classe
            //Statement qui permet d'écrire dans une base
            // nous voulons ici faire les deux mises à jour,
            // ou aucune des deux pour des raisons ( supposées )
            // d'intégrité de la base.

            select.executeUpdate("INSERT INTO COMPTE (NumCompte, Nom,
                Prenom, Solde)" +
                "VALUES (23, 'Grisolano', 'Philippe', 5550)");

            select.executeUpdate("INSERT INTO
                COMPTE (NumCompte,Nom,Prenom,Solde)" +
                "VALUES (22, 'Grisolano', 'Philippe', 3680)");

            //validation des deux requêtes si aucune des 2 n'a
            //générée une exception
            con.commit ();

            //fermeture de la requête et de la connexion
            select.close();
            con.close ();

        }
    }
}
```

```

catch(SQLException e)
{
    e.getSQLState ();
    try
    {
        //si la connection existe
        if (con!= null)
        {
            //on annule la transaction
            con.rollback();
        }
    }
    catch (Exception er) {}
}
}
}

```

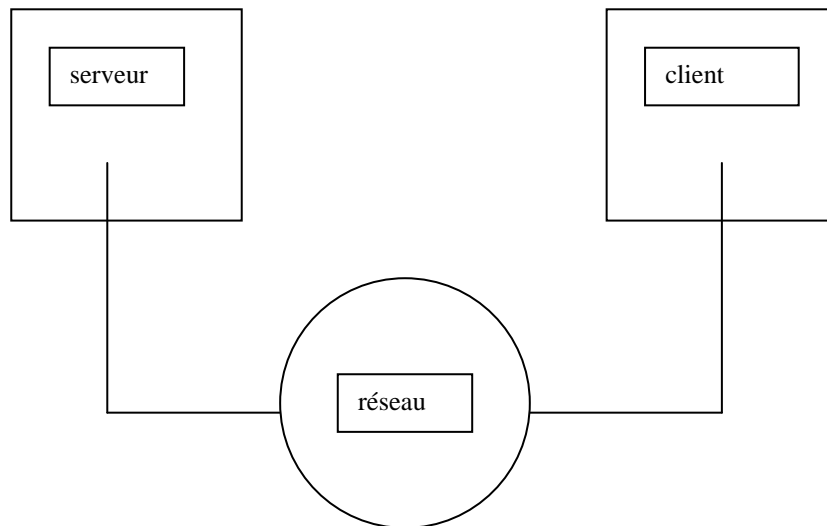
Exercice à réaliser :

Vous testerez les différentes requêtes en prenant des exemples sur vos bases de données. Vous testerez particulièrement les requêtes avec les procédures stockées les plus complexes (différents types de paramètres, et retour d'une valeur et d'un ResultSet), et également une transaction (ici sans procédure stockées, car alors c'est la procédure stockée qui fait le travail).

Vous choisirez, dans votre projet client serveur, un écran particulièrement riche, et je vous demande de le coder en java. L'analyse, et la conception de cet écran sont faites. Vous allez prouver que vous auriez pu réaliser votre projet client serveur en java.

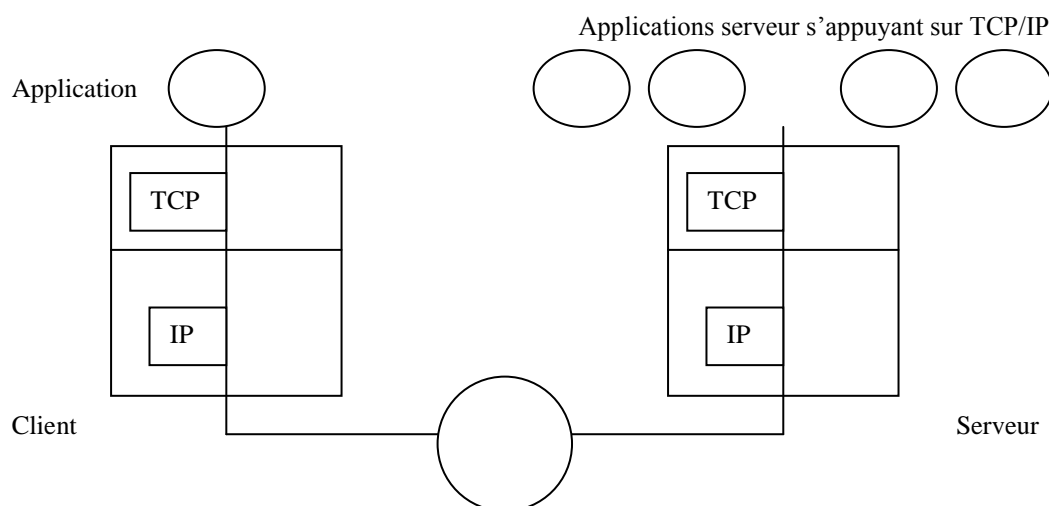
Cinquième partie : Threads et sockets

Nous allons réaliser une communication réseau du type client serveur entre deux applications java en utilisant les sockets.



Les bouts des supports de communication s'appellent des sockets.

Ce nom a été repris pour qualifier une technique de communication entre deux applications sur le réseau. Regardons ce qui se passe au niveau du logiciel pour une telle communication :



Quand le client sollicite le serveur il doit préciser quelle est l'application concernée. Pour cela il utilise un numéro d'identification de l'application (dit numéro de port) qui permet de faire le lien entre l'application cliente et le service demandé (application serveur).

Par exemple le numéro de port du service http est 80.

Par convention les numéros de port inférieurs à 1024 sont réservés à des services banalisés, les applications serveur spécifiques auront donc un numéro compris entre 1024 et 65535.

Bien sûr la machine cliente doit également donner l'identification de la machine serveur (par exemple son nom dans le domaine).

Le serveur enregistre son service sous un numéro de port, puis se met en attente d'un client par la méthode `accept()` de son instance de `ServerSocket`.

Le client peut alors établir une connexion avec le serveur en précisant le nom de la machine, et le numéro de port du service. Si la connexion est acceptée, il récupérera un socket vers le serveur dont il utilisera les flux en entrée et sortie pour communiquer avec le serveur.

Code du client :

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

public class Main
{
    public static void main(String[] args)
    {
        Socket c_socket;        // socket de communication avec le serveur
        DataOutputStream out;    // flux en sortie
        DataInputStream in;      // flux en entrée
        String rep ;            // chaine de communication

        try
        {
            //----Création de la connexion avec le serveur
            //----Bien renseigner le nom de la machine et
            //----numéro du port utilisé par le service
            c_socket = new Socket ("prof2407",1222);

            //----Création du flux de sortie
            out=new DataOutputStream (c_socket.getOutputStream());

            //----Envoi de l'information au serveur
            System.out.println("client : "+"envoi : coucou");
            out.writeUTF("coucou");

            //----Création du flux d'entrée
            in=new DataInputStream (c_socket.getInputStream());

            //----Lecture de la réponse du serveur
            rep=in.readUTF();
            System.out.println("client : reception : "+rep);
        }
        catch (IOException ioe)
        {
            System.out.println("client : la communication avec le serveur ne fonctionne pas");
        }
    }
}
```

Le serveur quand il aura accepté la connexion avec le client, récupérera un socket vers le client dont il utilisera les flux en entrée et sortie pour communiquer.

Code du serveur :

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

```

public class Main
{

    public static void main(String[] args)
    {
        Socket s;           // socket client
        ServerSocket s_server; // socket serveur
        String chaine;       // chaine de communication

        try
        {

            //----Etablissement du service
            s_server=new ServerSocket (1222);

            //----Tant que le serveur est sollicité
            while (true)
            {
                //----Attente du client
                s = s_server.accept();

                //----Création du flux d'entrée par rapport au socket
                DataInputStream in=new DataInputStream(s.getInputStream());

                //----récupération des données reçues
                chaine = in.readUTF();
                System.out.println("Serveur : reception" + chaine);

                //----Création du flux de sortie par rapport au socket
                DataOutputStream out= new DataOutputStream (s.getOutputStream());

                //----Envoi de la réponse au client
                System.out.println("Serveur : envoi bonjour" );
                out.writeUTF("bonjour ");
            }
        }
        catch (IOException ioe)
        {
            System.out.println("Serveur : erreur de fonctionnement");
        }
    }
}

```

Il est a noté que les envoi et réception d'information se font par des fonctions read et write UTF (c'est un codage de l'échange de données entre machines hétérogènes).

Dans le livre Java 1.1 vous trouverez un exemple de socket.

Travail à réaliser : vous réaliserez un serveur de calcul. Ce serveur retourne à ses clients le carré des nombres qui lui sont envoyés. Vous ferez des applications (et non des applets) avec une interface donnant pour le serveur ce qui est reçu, et la réponse renvoyée, et chez le client la saisie d'un entier, et l'affichage de son carré (renvoyé par le serveur bien entendu !!).

Les vrais serveurs sont multi threadés, le nôtre sera simple, en attendant la prochaine séquence.

Notion de Thread (ou processus léger)

Voir Java 1.1 p 528 et suivantes.

1) qu'est-ce qu'un thread ?

Les programmes réalisés jusqu'à maintenant font une chose unique, un traitement à la fois. Si nous voulons réaliser un programme permettant de travailler pendant qu'un travail long se déroule (chargement d'images, calcul long, ...) il nous faut construire des programmes qui fonctionnent en parallèle.

Ce parallélisme n'est qu'apparent si l'ordinateur qui exécute le programme n'a qu'un processeur. Néanmoins cette apparence est suffisante pour résoudre bien des problèmes.

De manière générale un programme qui s'exécute en parallèle d'autres programmes est appelé tâche ou processus. Un processus possède son code et ses données. Le code peut être commun à plusieurs processus, mais les données sont propres à chaque processus.

En java ces programmes sont des processus légers, ou threads. Ces processus sont dits légers car ils peuvent partager aussi leurs données. Cela leur permet de gagner en temps d'exécution (ce ne sera pas notre soucis ici !!).

2) création d'un thread.

```
Public class Monthread extends Thread           // Monthread est une classe qui hérite de Thread
{
    public void run()                           // méthode polymorphique lancée lors de l'exécution du
    {                                           // thread par start()
        while (true)
        {
            // faire quelque chose
        }
    }
    public static void main(String args)
    {
        Monthread mt = new Monthread() ;       // création d'un thread
        mt.start() ;                           // lancement de l'exécution du thread. La méthode start
                                                // lance la méthode run ( polymorphique ) du thread. D'autres
                                                // threads ou instructions pourraient être exécutés ici.
    }
}
```

Ici nous avons déclaré un thread et nous l'avons lancé.

Il est courant de créer un thread et de le lancer par une autre application. C'est presque la même chose :

```
Public class Monthread extends Thread           // Monthread est une classe qui hérite de Thread
{
    public void run()                           // méthode polymorphique lancée lors de l'exécution du
    {                                           // thread
        while (true)
        {
            // faire quelque chose
        }
    }
}

Public class essai
{
    public static void main(String args)
    {
        Monthread mt = new Monthread() ;       // création d'un thread
        mt.start() ;                           // lancement de l'exécution du thread. La méthode start
```

```

// la méthode run ( polymorphe ) du thread. D'autres
// threads ou instructions pourraient être exécutés ici.
    }
}

```

Il existe une autre manière de créer un thread. Une classe java ne peut hériter que d'une seule classe. Comment faire pour créer une classe héritant d'une classe et de thread en même temps ?

Pour cela nous allons utiliser les interfaces. Une interface est une description de méthode nécessaire pour qu'un service soit rendu. L'interface Runnable dit qu'il faut qu'une classe possède une méthode publique void qui s'appelle run et sans paramètre, pour qu'on puisse en faire un thread. Voici un exemple :

```

Public class Renable extends classe mère implements Runnable
{
    public void run()          // méthode dont la présence est rendue nécessaire par l'interface Runnable
    {
        // ce qu'on désire faire
    }
}

```

puis dans une classe d'application par exemple :

```

Renable r = new Renable();      // on crée un objet r de la classe renable
Thread t = new Thread(r);      // on crée un thread à partir de l'objet r
t.start();                     // on démarre le thread, en fait on lance la méthode run de r.

```

3) Les méthodes de la classe Thread et leur évolution.

Voici quelques méthodes de la classe Thread dont l'usage est fréquent. Vous pouvez consulter l'aide pour avoir les autres méthodes.

int [getPriority\(\)](#)

Retourne la priorité du thread (de 1 à 10, 10 étant la priorité la plus forte).

Boolean [isDaemon\(\)](#)

test si ce thread est un thread démon (thread activé en tâche de fond, ou cycliquement)

void [join\(\)](#)

attend la mort de ce thread.

void [resume\(\)](#)

Cette méthode est fortement déconseillée.

Au vu des problèmes d'étreinte fatale qu'elle engendre, le programmeur est fermement invité à ne plus utiliser cette méthode. Elle permet de réveiller un thread endormi par la méthode suspend().

void [run\(\)](#)

Cette méthode sera appelée au démarrage du thread (par la méthode start()).

void [setDaemon\(boolean on\)](#)

définit si le thread est un thread démon ou un thread utilisateur.

void [setPriority\(int newPriority\)](#)

change la priorité du thread (1 moins prioritaire, 10 plus prioritaire).

void [sleep\(long millis\)](#)

endort le thread pour un certain nombre de millisecondes. (cela permet à d'autres process de prendre la main).

void [start\(\)](#)

démarre l'exécution du thread en appelant la méthode run().

void [stop\(\)](#)

Cette méthode est fortement déconseillée.

Au vu des problèmes d'étreinte fatale qu'elle engendre, le programmeur est fermement invité à ne plus utiliser cette méthode. Elle permet d'arrêter un thread.

void [suspend\(\)](#)

Cette méthode est fortement déconseillée.

Au vu des problèmes d'étreinte fatale qu'elle engendre, le programmeur est fermement invité à ne plus utiliser cette méthode. Elle permet d'endormir un thread. Le Thread sera réveillé par la méthode resume().

void [yield](#)()

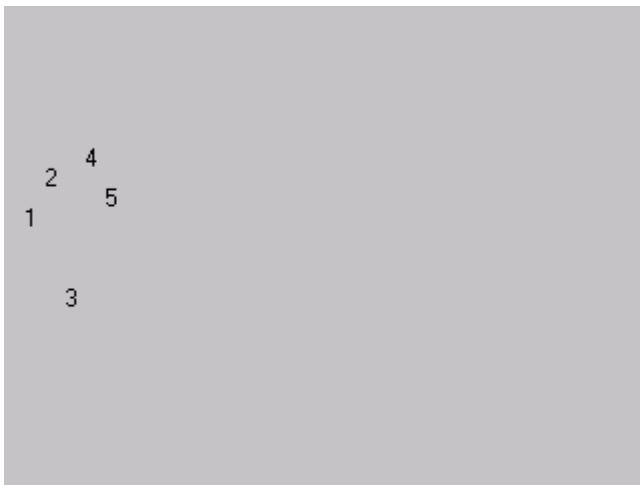
permet de faire une pause à un Thread. Un autre Thread de même priorité sera alors actif s'il y en a un. Sinon le Thread redémarre. C'est un partage coopératif du processeur entre threads de même priorité.

Exercice 1 :

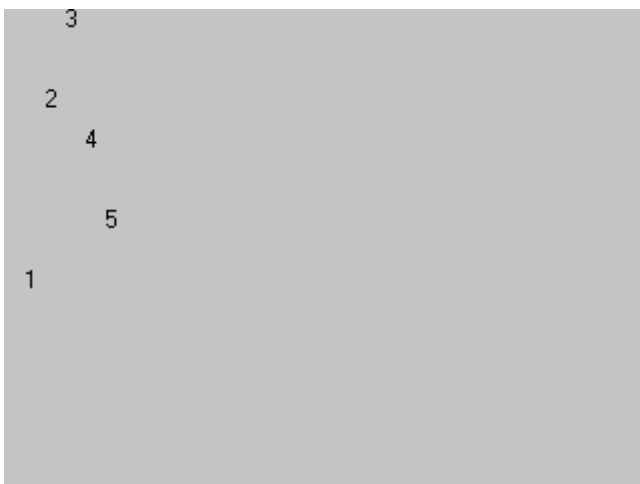
Nous allons mettre en évidence de manière simple ce principe de thread. Une Frame affichera le défilement cyclique vertical de plusieurs chiffres. Chaque chiffre représente un thread, qui a un délai de réveil cyclique (différent pour chaque thread ce qui permet de simuler des vitesses différentes, et de l'ordre de 1000 à 2500 millisecondes) et qui a chaque réveil descend de dix pixels. Arrivé en bas de la zone d'affichage de la Frame elle remonte en haut de cette zone.

Voici deux vues d'exécution de ces threads. Ici il a été pris cinq threads, mais il ne coûterait rien d'en prendre dix. C'est la même classe qui a été instanciée plusieurs fois.

Au début de l'exécution :



Un petit peu plus tard (après quelques tours) :



Le code d'une solution est fourni en annexe (ce n'est pas le code dont l'image est issue), le travail minimum est de faire fonctionner ce code. Vous pouvez également chercher une solution par vous-même.

```

// classe définissant la fenêtre

import java.awt.Frame;
import java.awt.event.*;

public class MovingNumber extends Frame
{
    // fenêtre de visualisation des labels
    public MovingNumber()
    {
        // fermeture de la fenêtre
        winClose wc = new winClose();
        addWindowListener(wc);
        // pas de layout pour que les labels puissent bouger librement
        setLayout(null);
        setSize(200,600);
        setVisible(true);
    }
    class winClose extends WindowAdapter
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    }
}

// classe principale de l'application
import java.awt.Color;

public class TestThreads {

    public static void main(String[] args)
    {
        MovingNumber mn = new MovingNumber(); // création de la fenêtre
        // création des 7 threads avec leurs paramètres
        ThreadNumber tn1 = new ThreadNumber("1",50,30,Color.orange,100);
        ThreadNumber tn2 = new ThreadNumber("2",70,30,Color.red,150);
        ThreadNumber tn3 = new ThreadNumber("3",90,30,Color.magenta,200);
        ThreadNumber tn4 = new ThreadNumber("4",110,30,Color.cyan,110);
        ThreadNumber tn5 = new ThreadNumber("5",130,30,Color.pink,250);
        ThreadNumber tn6 = new ThreadNumber("6",150,30,Color.green,230);
        ThreadNumber tn7 = new ThreadNumber("7",170,30,Color.blue,120);

        // les threads sont des labels qui bougent. Il faut donc les ajouter à la fenêtre
        mn.add(tn1);
        mn.add(tn2);
        mn.add(tn3);
        mn.add(tn4);
        mn.add(tn5);
        mn.add(tn6);
        mn.add(tn7);

        // Démarrage des threads ( mettre un moteur dans chaque label et démarrer )
        new Thread(tn1).start();
        new Thread(tn2).start();
        new Thread(tn3).start();
        new Thread(tn4).start();
        new Thread(tn5).start();
        new Thread(tn6).start();
    }
}

```

```

        new Thread(tn7).start();
    }
}

// classe de gestion d'un process
import java.awt.*;

// chaque process est un label qui a un moteur qui le fait bouger
public class ThreadNumber extends Label implements Runnable
{
    private int m_lx;                // coordonnée en x
    private int m_ly;                // coordonnée en y
    private int m_dort;              // temps de sommeil
    private boolean versBas = true;  // sens de déplacement

    public ThreadNumber(String lab,int lx,int ly,Color c, int dodo)
    {
        super(lab);
        m_lx = lx;
        m_ly = ly;
        m_dort = dodo;

        setSize(10,10);             // taille du label
        setBackground (c);           // couleur du label
        setLocation(lx,ly);          // positionnement du label
    }

    public void dort(long time)
    {
        try
        {
            Thread.sleep(time);
        }
        catch (InterruptedException ue)
        {
            System.out.println(ue.getMessage());
        }
    }

    public void run()
    {
        while(true)                // toujours
        {
            if(versBas)
            { // le label descend
                if(m_ly < 400)
                { // la coordonnée y est incrémentée
                    m_ly++;
                    setLocation(m_lx,m_ly);
                }
            }
            else
            { // changement de sens
                versBas = false;
            }
        }
        else
        { // le label monte
            if(m_ly > 30)
            { // la coordonnée y est décréementée

```



```

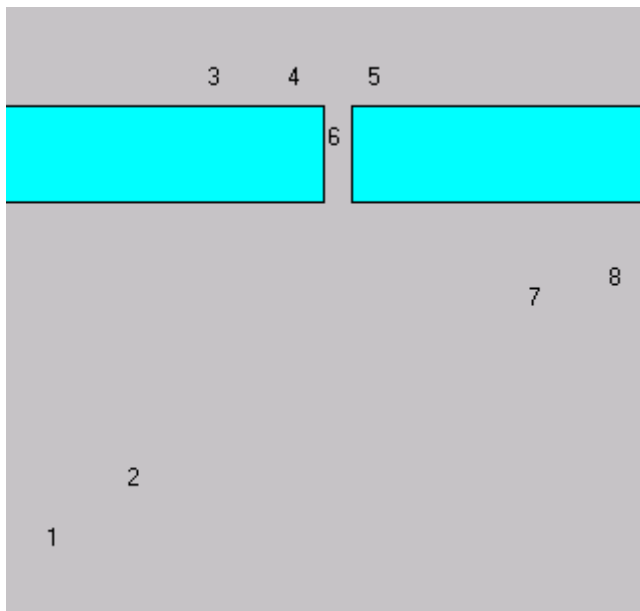
        m_ly--;
        setLocation(m_lx,m_ly);
    }
    else
    {
        // changement de sens
        versBas = true;
    }
}
dort(m_dort);
}
}
}

```

Exercice 2 :

Nous allons reprendre le même problème, mais les compilées pour aller de haut en bas doivent traverser un pont fragile au dessus d'une rivière remplie de piranhas et sur lequel ne peut passer qu'un thread à la fois. Chaque thread aura une priorité différente.

Regardons ce que cela donne visuellement :



Les compilées 3,4 et 5 sont en attente que le pont soit libre. Les autres continuent leur route normalement.

Ici les compilées se partagent une ressource commune qui est le pont. Quand ils arrivent au pont, Ils auront un algorithme du type :

Si le pont est libre alors

Je réserve le pont

J'avance d'un pas sur le pont

Sinon

Je ne fais rien

Je m'endors de mon temps de sommeil

Regardons ce qui se passe si le thread un exécute cet algorithme au moment ou le thread 2, plus prioritaire, se réveille.

Thread 1

```
Si le pont est libre      // oui, à ce moment il est libre
                        // hélas se réveille le thread 2
```

Thread2

```
Si le pont est libre // oui, à ce moment il est libre
    Je réserve le pont
    J'avance d'un pas sur le pont
    Je m'endors de mon temps de sommeil
```

```
                        // le thread 1 reprend la main
    Je réserve le pont
    J'avance d'un pas sur le pont
    Je m'endors de mon temps de sommeil
```

Résultat il y a deux compilées en même temps sur le pont. Il est donc nécessaire d'utiliser un mécanisme de sémaphore. Ici il faudra aller voir la notion de **synchronized** pour résoudre ce problème particulier (voir Java 1.1 page 529 et suivantes). Ici il semble bon de rendre indivisible l'accès au pont en rendant cette fonction synchronized.

- Vous allez d'abord tester votre application sans méthode synchronized. Elle semble bien fonctionner. Cela tient plus à la chance qu'à vos prouesses, aussi grandes soient elles.
- Vous rajouter un endormissement de 500 millisecondes entre le test qui vérifie si le pont est libre, et le positionnement du pont à occupé, et vous relancez votre application. Plusieurs threads traversent le pont en même temps (vous venez de forcer un « mauvais » aléas).
- Vous synchronisez correctement votre accès au pont, et votre application fonctionne à nouveau, malgré le mauvais sort que vous avez forcé.
- Vous pouvez maintenant enlever votre endormissement au plus mauvais moment de 500 millisecondes.

La méthode join() permet à un thread d'attendre la mort d'un autre thread. Elle est très intéressante, mais nous n'aurons pas le temps de l'approfondir ici. Les méthodes notify, notifyall et wait (qui viennent de l'héritage de Object) permettent de compléter cette palette d'outils pour gérer les problèmes du multitâche.

Vous réaliserez cette application, en faisant bien attention aux problèmes de synchronisation entre les process.

Exercice 3 :

Nous allons reprendre le problème de notre serveur de carré de nombre. Tel qu'il a été fait, ce serveur marche en mode déconnecté : chaque client qui veut un service du serveur se connecte, demande le service (et un seul service rapide) ici le carré d'un nombre, puis se déconnecte.

Ce type de serveur correspond aux problèmes où le client et le serveur n'ont pas de suivi de leurs relations. Auquel cas il va falloir travailler en mode connecté. Ce que nous allons faire avec notre serveur socket : Le client se connecte, puis demande un carré. Le serveur lui répond, ainsi de suite jusqu'à ce que le client demande le carré de zéro, le serveur lui répond, puis interrompt la communication avec le client. Le client pour sa part s'arrête quand il a reçu la réponse au carré de zéro. Pendant tout cet échange, le client et le serveur étaient connectés.

Nous allons coder un tel client et serveur. Il devient évident qu'un autre client qui veut se connecter devra attendre que le premier client ait fini.

Dans un deuxième temps, grâce aux threads, nous allons pouvoir traiter plusieurs clients en mode connecté. Le serveur reçoit un client, et le sous-traite à un thread qui traite la connexion avec le client jusqu'au bout. Le serveur pendant ce temps attend un nouveau client qu'il sous traitera à nouveau à un thread. Nous aurons ici réalisé un serveur « multi threadé ».

Exercice 4 :

Nous allons reprendre notre application de supervision. La supervision reçoit les messages et alarmes venant de différents dispositifs de mesure, et de sécurité. Le Panel de simulation de création d'alarmes et de messages est donc à enlever de cette simulation.

Nous allons simuler les créations de message et d'alarmes sur différents points du réseau. La supervision est chargée de collecter les informations sur ces messages et alarmes (il y a donc un serveur socket caché derrière cela), puis à partir de ces informations de créer les messages et alarmes, et de les mettre dans l'historique. Le panel de simulation de création de messages et alarmes devient alors un client socket qui simule maintenant la création de message et d'alarme en différents points du réseau.

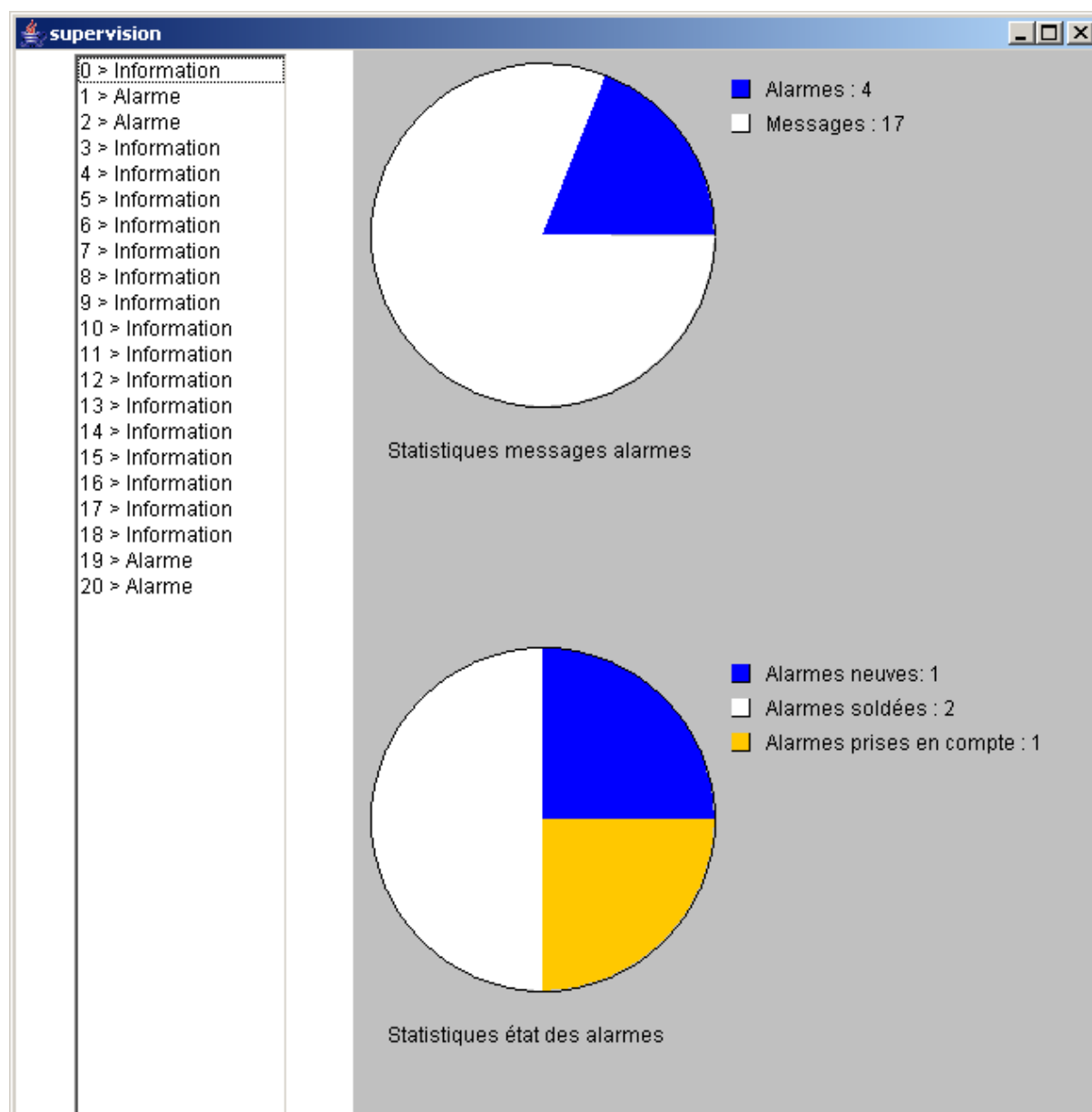
Exemple d'écran de simulation de la création d'une alarme ou d'un message sur le réseau.

The image shows a software window titled "simulation création événement". It contains four input fields and a button:

- Type :** A dropdown menu with "Alarme" selected.
- Libellé :** A text box containing "incendie".
- Source événement :** A text box containing "salle machine".
- Code de gravité :** A dropdown menu with "1" selected.
- Send**: A large button at the bottom left.

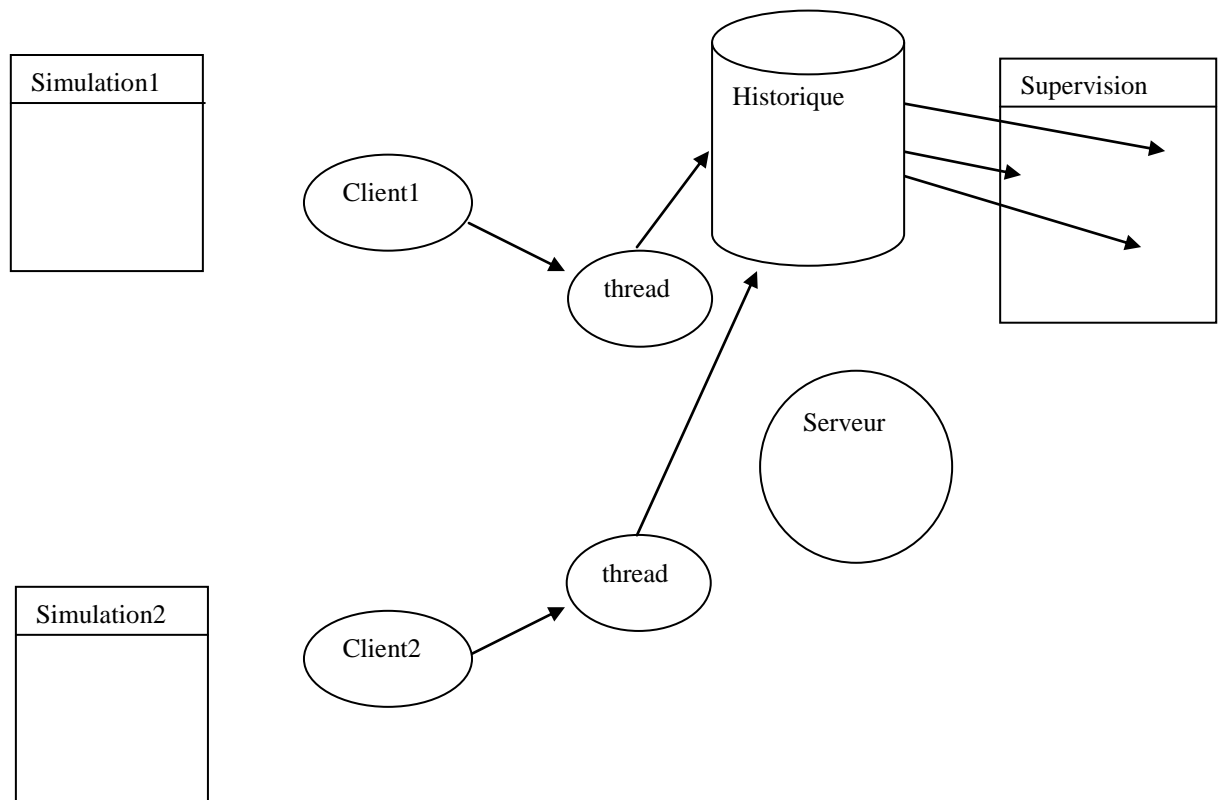
Cette application pouvant tourner sur n'importe quelle machine du réseau de l'entreprise.

Exemple d'écran de la supervision :



Cette application tournant sur l'ordinateur de supervision.

Voici un schéma du fonctionnement attendu pour notre supervision maintenant :



Chaque thread va mettre à jour l'historique, et l'historique prévient toutes les vues de se mettre à jour.

Sixième partie : Réalisation des interfaces java avec les classes SWING

1) introduction

Nous avons utilisé les classes du package AWT pour créer les interfaces graphiques de nos applications. Comment fonctionnent ces classes graphiques ?

Chaque objet d'interface est associé avec un objet peer :

Button ← → ButtonPeer

Objet AWT objet lié à l'API de l'operating system de la machine d'exécution.

Notre objet, quand il s'active, déroule du code différent selon le système d'exploitation de la machine sur laquelle il s'exécute. Ceci implique que nos programmes auront des interfaces différentes, voire des comportements différents suivant la machine sur laquelle ils s'exécutent.

Cela explique également que les classes AWT soient peu nombreuses, avec des possibilités peu étendues. En effet, les objets AWT représentent les possibilités que l'on retrouve dans tous les systèmes d'exploitation, c'est donc le plus petit dénominateur commun aux API de ces systèmes d'exploitation, en terme de comportement graphique.

Une solution pour résoudre ces deux problèmes (pas exactement le même comportement suivant la machine hôte, et une interface graphique pauvre) : les objets doivent être réécrits en java, sans s'appuyer sur les API des machines hôtes.

Ces composants java, dits composants légers (en opposition aux composants lourds, ceux qui font appel à l'API de la machine hôte) vont avoir le même comportement partout. D'autre part, nous allons pouvoir disposer de tous les composants nécessaires, sans limitation.

Ces composants forment les classes Swing.

Tous les composants ne peuvent pas être légers. La fenêtre principale d'une application est en interaction avec le système d'exploitation de la machine hôte. Cette fenêtre ne peut donc pas être un composant léger. Par contre, tout ce qu'elle contient va être léger.

Quand un composant Swing redéfinit un composant AWT, il prend le même nom que lui, en ajoutant un J devant (JButton pour Button, ...). En composants lourds en AWT, il reste JApplet, JWindow, JFrame, JDialog. En un mot toutes les classes qui permettent de créer une fenêtre gérée par le système d'exploitation de la machine hôte. Ces composants héritent de leurs homologues AWT, avec enrichissement.

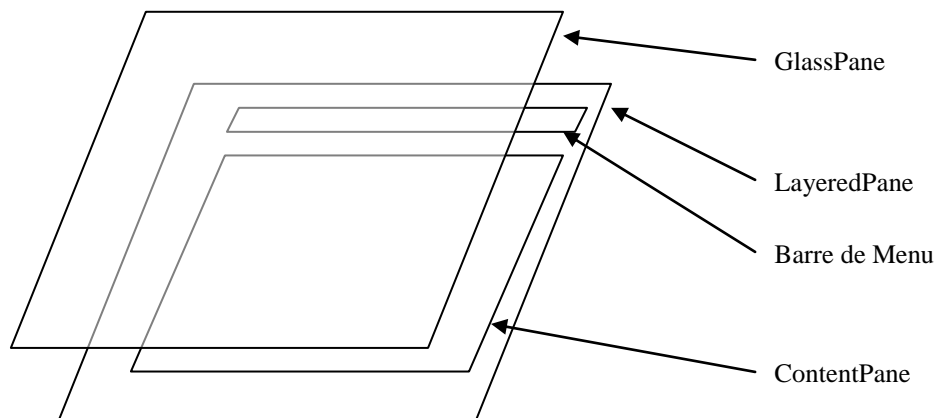
Nous pouvons choisir l'apparence des composants (sauf pour les lourds), les composants ayant des apparences différentes pour Windows, Motif, ou Internet (Métal). Swing donne donc la possibilité de choisir le look and feel de ses composants, et même d'en changer en cours d'exécution.

Nous allons avoir une plus grande richesse de composants (JSlider, JTree, JTable, ...) et pour chaque composant, une plus grande variété de comportements (images sur les boutons, images et texte dans une liste, ...).

2) Comment est conçu un container

Nous allons regarder comment est conçu un container, à travers le fonctionnement d'une JFrame.

Une JFrame contient un RootPane constitué ainsi :



Vu du RootPane d'une JFrame

Le GlassPane est la vitre de protection de la fenêtre. Si elle est activée, elle peut capter les événements qui se produisent sur la fenêtre, et ainsi ne rien laisser passer sur les couches basses. C'est le principe des Splash Screen (image apparaissant pendant le chargement et l'initialisation d'une application, empêchant l'utilisation de celle-ci pendant l'initialisation). Généralement la GlassPane est transparente.

Le LayeredPane est une succession de panneaux transparents superposés intercalés entre la GlassPane et la ContentPane (panel contenant les composants de la fenêtre). La couche du bas est affichée la première par dessus la ContentPane, et ainsi de suite pour toutes les couches. Cela nous permet de gérer les différentes superposition que l'on trouve dans une application fenêtrée (menus contextuels, barres d'outils flottante, glisser déplacer, ...).

Il y a des couches prédéfinies, dans l'ordre :

DragLayer	(symboles du drag , c'est le plus haut niveau)
PopupLayer	(popup et tool tips)
ModalLayer	(fenêtres modales)
PaletteLayer	(barres d'outils flottantes ou palettes)
DefaultLayer	(panneau général)
ContentLayer	(ContentPane et barre de menu)

De tout cela nous allons retenir une chose principale : c'est dans le `ContentPane` de la `JFrame` que l'on ajoute les composants définissant notre fenêtre. Les autres éléments seront mis là où il faut par défaut.

Le `add` ne se fait plus sur la fenêtre, mais sur le `ContentPane` de la fenêtre. Nous obtenons ce `ContentPane` par la méthode `getContentPane()`. D'où les commandes :

`getContentPane().add(new JButton());`

N'oubliez pas de travailler avec des composants Swing (commençant par J).

Evitez de mélanger dans une même fenêtre les composants Swing et les composants AWT, cela donne parfois des comportements aussi bizarres qu'inexplicables.

3) Quelques exemples simples :

Si la technique pour passer d'un programme utilisant AWT à un programme utilisant les Swing semble simple (nous verrons plus loin que pour les listes ce n'est pas si simple qu'il n'y paraît), nous allons toutefois parcourir quelques exemples quelques enrichissements des classes Swing par rapport aux classes AWT (notez que ces exemples ne cherchent pas à être des applications, mais à bien montrer le fonctionnement de certains composants).

- couleur : ici nous utilisons un `JColorChooser` pour choisir une couleur. Maintenant, pour demander une couleur à votre utilisateur, vous ne pouvez plus faire moins bien que cela.
- edith : un petit éditeur qui a pour but de montrer le fonctionnement d'un `JFileChooser` (boîte de dialogue de choix d'un fichier). Ce composant doit se construire, ou se paramétrer avec un `FileFilter` (équivalent à « tous les fichiers *.abc ») et utiliser une classe utils pour gérer les extensions de fichier. Cet exemple permet également de gérer une boîte à bouton dockable.
- Essai2D : une petite porte ouverte vers les possibilités graphiques 2D
- Internewin : gestion de fenêtres internes qui sont des composants légers.
- `OptionPane` : toutes les boîtes de dialogue dont vous avez rêvé, et même plus (ici les `OptionPane` ne sont pas internes car les `OptionPane` internes ne sont pas des fenêtres modales, je m'en suis donc tenu aux `OptionPanes` externes).
- Split : logique de partage d'une fenêtre en deux panels.
- Slider : utilisation de sliders (composant `JSlider`).

Et ce n'est bien sûr qu'un début ...

Faites fonctionner les exemples et jetez un œil sur le code.

4) Professionnalisons notre développement

Nous allons voir deux aspects du développement de logiciels à forte composante IHM. La première est la possibilité de faire la même action de plusieurs manières différentes, la deuxième est l'internationalisation des applications.

- Regroupement d'actions identiques : Dans une application professionnelle, nous trouvons souvent la même action accessible par les menus, par une boîte à bouton, par un menu contextuel, etc ... Il apparaît clairement qu'il ne serait pas professionnel de coder séparément chacun de ces composants (car ils effectuent le même traitement). De

même, si je désire désactiver le comportement momentanément, il est délicat de gérer l'ensemble de ces composants séparément (encore faut-il avoir la connaissance des autres composants rendant le même service, au moment où nous codons notre composant).

Nous allons pour cela utiliser la classe action. Cette action est l'âme du comportement que l'on veut décrire. Quand nous voudrions créer un composant sur ce comportement, nous lui donnerons en paramètre cet action, ainsi tous les composants attachés à un même comportement seront liés entre eux, et il suffit alors d'en griser un pour que tous les autres soient grisés aussi.

Exemples :

- menu : gestion d'un menu et d'une boîte de boutons, les éléments du menu étant repris dans la boîte de boutons. Cet exemple se marierait très bien avec l'application edith vue plus haut. Cet exemple nous montre également la possibilité qui nous est donnée de mettre un contour aux composants (notion de border).
- Popup : ici aussi gestion des actions entre un menu, un bouton, et un menu contextuel.
- internationalisation :
Les logiciels doivent aujourd'hui s'adapter à la langue des différents pays, mais aussi aux habitudes de notation comme les dates, l'écriture des nombres, ...
Exemple :
 - bundle : chaque contextualisation du logiciel pour un pays redéfinit les locales. Une locale est un couple (langue, pays) qui permet de définir le ressource bundle qui permet de trouver la chaîne de caractère dans la langue adéquate (à partir de fichier properties), mais permet aussi de mettre en place des formats pour écrire les dates, les nombres, etc ...
- quelques ajouts de java6
 - Création d'un SplashScreen : la visualisation du résultat est obtenue en double cliquant sur le fichier Splash.jar du répertoire java swing. Approfondir ici la notion de SplashScreen, la notion de manifeste, et la création d'un jar d'exécution (contenant le manifeste (manif.mf) les .class, et les ressources nécessaires (ici une image .png).
 - Gestion des icônes : voir projet Tray : changement de l'icône de la fenêtre, mise en place d'une icône dans la barre des tâches, info bulle sur cette icône, et gestion d'un menu contextuel sur cet icône.

5) définition de l'architecture Modèle-Vue-Contrôleur (dite MVC)

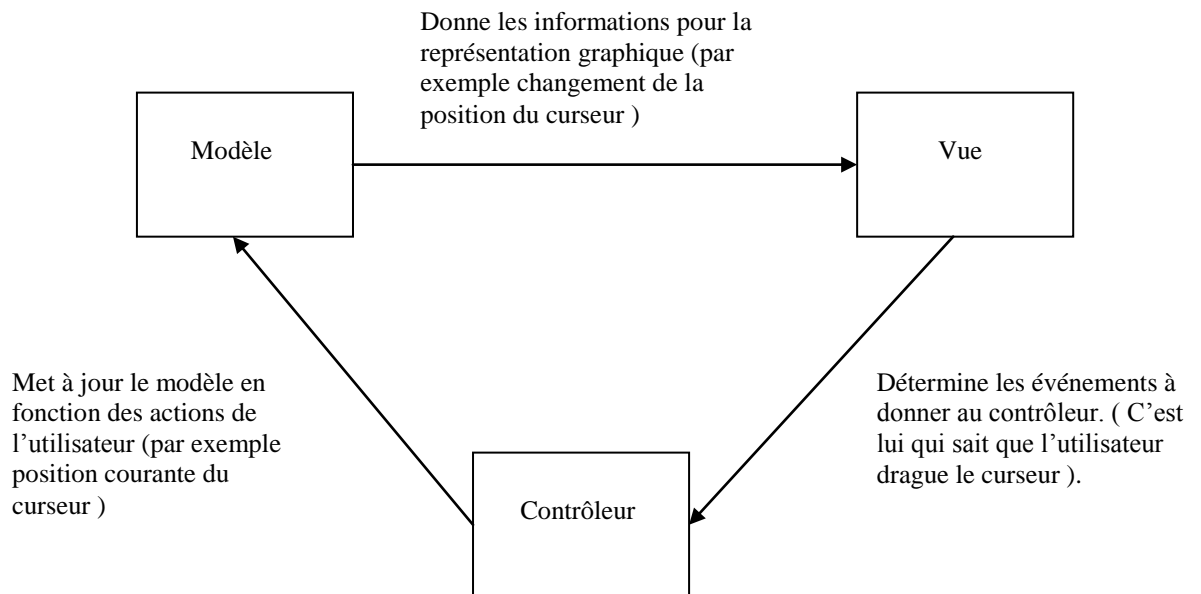
Les composants graphiques devraient appliquer l'architecture MVC. Cela veut dire que chaque composant graphique devrait être composé de trois entités :

Le Modèle : il contient toutes les données qui représentent l'état du composant : par exemple pour un slider le modèle contiendrait le minimum, le maximum, et la valeur courante.

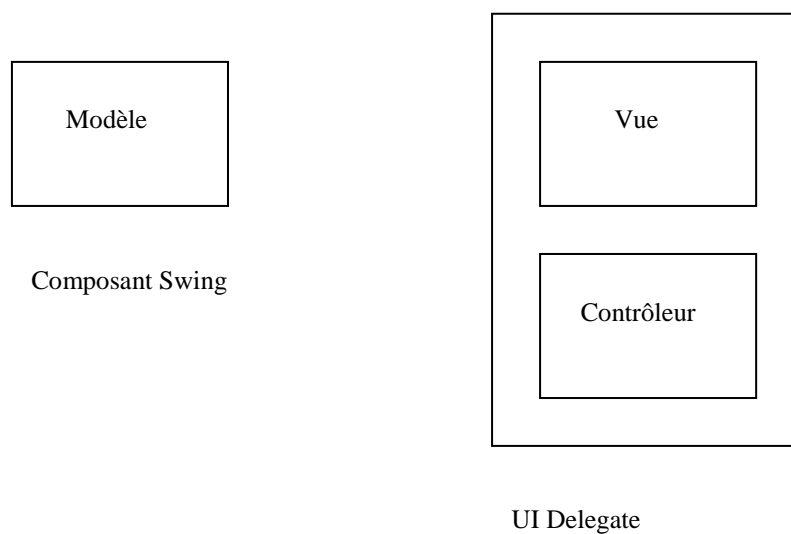
La Vue : cela fait référence à comment le composant est dessiné sur l'écran. Nous remarquons que suivant le type des systèmes d'exploitation, les composants peuvent avoir des représentations différentes. La vue caractérise donc comment notre composant sera dessiné à l'écran, dans ses différents états (bouton enfoncé, ...).

Le Contrôleur : c'est la logique de fonctionnement du composant, face aux actions de l'utilisateur. C'est le contrôleur qui définit si le composant réagit aux clics souris, à la touche Enter, à la perte du focus, ... Pour notre slider, c'est le contrôleur qui décide qu'il est réactif aux clics sur ses bornes, et aux événements drag sur son curseur.

Bien sûr ces trois entités collaborent :



Les composants Swing suivent le pattern MVC. Le découpage des composants a été réalisé ainsi :



Les vue et contrôleur ont été réunis en un composant (nommé UI Delegate), ce qui permet de manipuler en une seule fois ce qui différencie les LookAndFeel différents (vue et comportement qui sont différents suivant les plateformes que l'on veut simuler).

Ainsi pour changer l'apparence d'un composant, il suffit de changer son UI Delegate. >Cela peut se faire en cours d'exécution de l'application par un code comme ceci :

```
UIManager.setLookAndFeel(« .... ») ; // mise en place du LookAndFeel choisi  
SwingUtilities.updateComposantTreeUI(this) ; // this représentant la fenêtre principale
```

L'autre aspect de la chose est le travail sur le modèle. Une modification des valeurs du modèle entraîne une mise à jour du composant. Ainsi nous allons beaucoup travailler sur les modèles pour de nombreux objets (surtout sur les modèles par défaut, sans nous en rendre compte, car c'est transparent pour un grand nombre de composants). Le UI Delegate a un écouteur sur le modèle, et sera mis à jour automatiquement quand le modèle de donnée sera modifié.

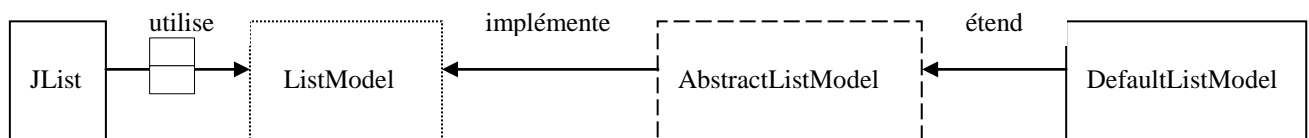
Nous allons approfondir deux composants, pour lesquels nous allons travailler explicitement sur le modèle de donnée, voire nous allons créer le modèle de données. Ces composants sont le JList, et le JTree. Un troisième composant serait particulièrement intéressant à développer, c'est le JTable. Les approfondissements réalisés sur le JTree vous permettront d'aborder le JTable assez sereinement.

6) JList

- 1) Nous désirons créer une liste avec un nombre fixe et invariable d'éléments. Le constructeur de JList nous permet de le faire, nous utilisons un modèle par défaut, cela est transparent pour nous.

Voir l'exemple liste1

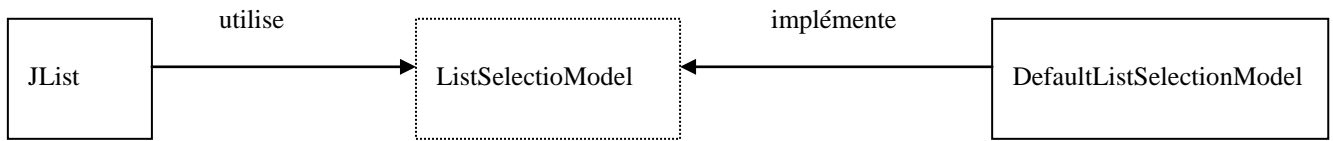
- 2) Nous désirons créer une liste avec un nombre d'éléments variable, et la possibilité d'ajouter ou d'enlever des éléments de la liste. Il faut alors définir un modèle de donnée. Java propose un modèle de donnée par défaut (DefaultListModel). Celui-ci repose sur un vecteur. Si il ne nous convient pas vous pourriez vous même redéfinir votre modèle, en implémentant l'interface ListModel, ou en étendant la classe abstraite AbstractListModel.



Dans le cas général, le modèle par défaut nous convient très bien. Voir l'exemple liste2.

- 3) Nous désirons maintenant nous intéresser à la sélection d'un élément de la liste. Si nous voulons changer le modèle de sélection, nous pouvons soit récupérer le modèle de sélection, et le changer (c'est le choix de l'exemple). Nous pourrions également

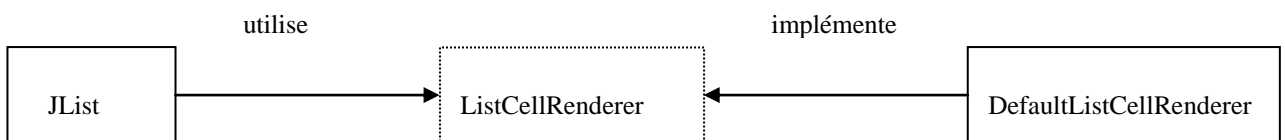
créer un modèle de sélection par défaut, et l'associer à la liste, soit recréer une classe décrivant du modèle de sélection, et implémentant le ListSelectionModel.



L'architecture des classes est ici très importante pour comprendre comment ces composants doivent être manipulés, même si nous utilisons souvent les modèles par défaut. Les relations entre les classes seront souvent simplifiées.

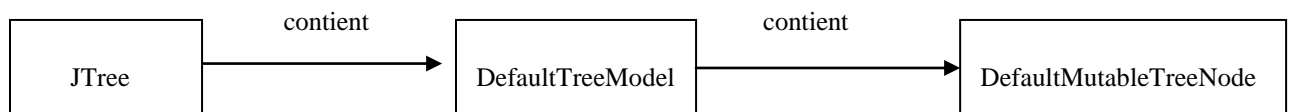
Voir l'exemple liste3.

- 4) Apparence du contenu de la liste. Nous allons créer une classe de rendu de cellule pour associer un icône avec le texte de chaque choix de la liste. Nous créons cette classe à partir de l'interface ListCellRenderer. Voir l'exemple liste4.



7) JTree

1) Sur le JTree le même travail d'investigation a été mené. Le composant JTree est un composant permettant de représenter les informations arborescentes. Le JTree travaille avec un modèle d'arbre par défaut le DefaultTreeModel. Ce modèle contient des descriptions des nœuds de l'arbre, les TreeNode. TreeNode est une interface décrivant les services rendus par un nœud d'une arborescence. Le Nœud le plus simple (contenant du texte, et sans comportement notoire) est le DefaultMutableTreeNode qui implémente l'interface TreeNode.

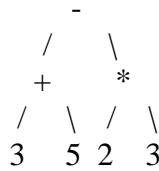


Voir l'exemple TreeSimple

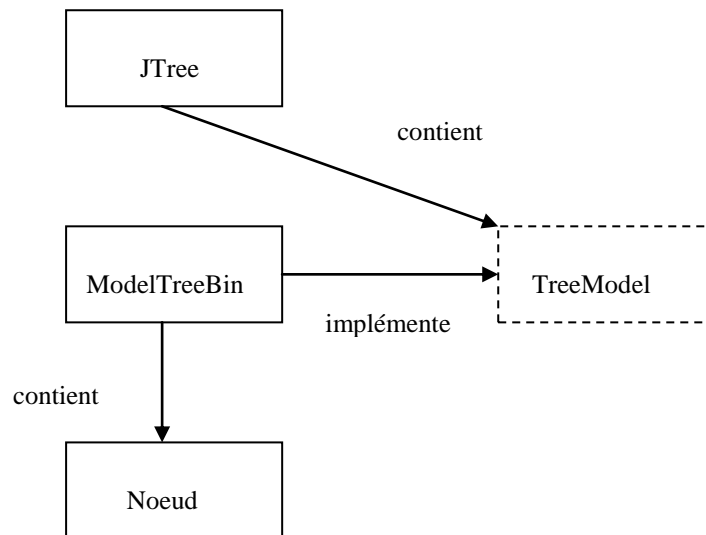
- 2) Dans l'exemple suivant nous rajoutons un écouteur sur l'arborescence (TreeSelectionListener) pour savoir si la sélection a changé (valueChanged). Voir l'exemple TreeSélection.
- 3) Dans l'exemple suivant nous avons rendu l'arborescence éditable. Exemple TreeEditable.
- 4) Enfin sur ce même exemple nous avons rajouté les tooltips quand nous parcourons les feuilles et les nœuds de l'arbre (avec des tooltips différents dans ces deux cas). Ceci implique d'ajouter à l'arbre un comportement getToolTipText, et d'enregistrer l'arbre au niveau du ToolTipManager. Voir l'exemple TreeToolTip.

- 5) Nous allons revenir sur la structure de l'arbre, en prenant un autre exemple, pour sortir des classes de modèle et de nœud par défaut. Notre arbre sera formé à partir d'une expression arithmétique complètement parenthésée.

$((3+5)-(2*3))$ donnera



Notre modèle de données étant spécifique, nous allons le construire en implémentant l'interface `TreeModel`. Ici nous avons choisi de construire les nœuds indépendamment de l'interface `TreeNode`, car nous ne travaillons pas à partir du modèle de données par défaut.



Voir l'exemple `TreeBin5`.

- 6) Nous voulons écouter les ouvertures des sous arbres, pour écrire dans le nœud la sous expression si le sous arbre est fermé, mais l'opérateur seulement si le sous arbre est ouvert. Pour cela nous mettons en place un écouteur `TreeExpansionListener`. Voir l'exemple `TreeBin6`.

- 7) Nous voulons rendre l'arbre modifiable. N'importe quel nœud ou feuille peut être modifié. Une feuille peut être remplacée par un nouveau sous arbre. La valeur de l'expression est calculée, et réévaluée si il y a modification. L'écriture de l'expression parenthésée est également mise à jour. Pour cela nous avons mis en place un écouteur sur le modèle de donnée (`TreeModelListener`), pour remettre à jour les différents champs. L'arborescence est mise à jour automatiquement à partir du modèle des données. Voir l'exemple `TreeBin7`.

- 8) Nous avons ici réuni deux fonctionnalités nouvelles.

- 8) nous avons mis en place un rendu de cellule, pour pouvoir donner à notre arborescence le design voulu. Pour cela nous avons créé la classe RenduCelluleArbre qui est un TreeCellRenderer.
- 9) Nous avons mis en place un éditeur de cellule spécial pour les nœuds ouverts (actif sur le double click gauche sur la cellule désirée). La modification consiste alors à proposer de modifier un opérateur ; Nous utilisons pour cela un Combobox. Ceci nous oblige alors à créer un éditeur pour les autres cellules, qui est un éditeur de texte, et de créer un éditeur unique pour l'ensemble des cellules qui nous délivrera l'éditeur adéquat.

D'où les nouvelles classes :

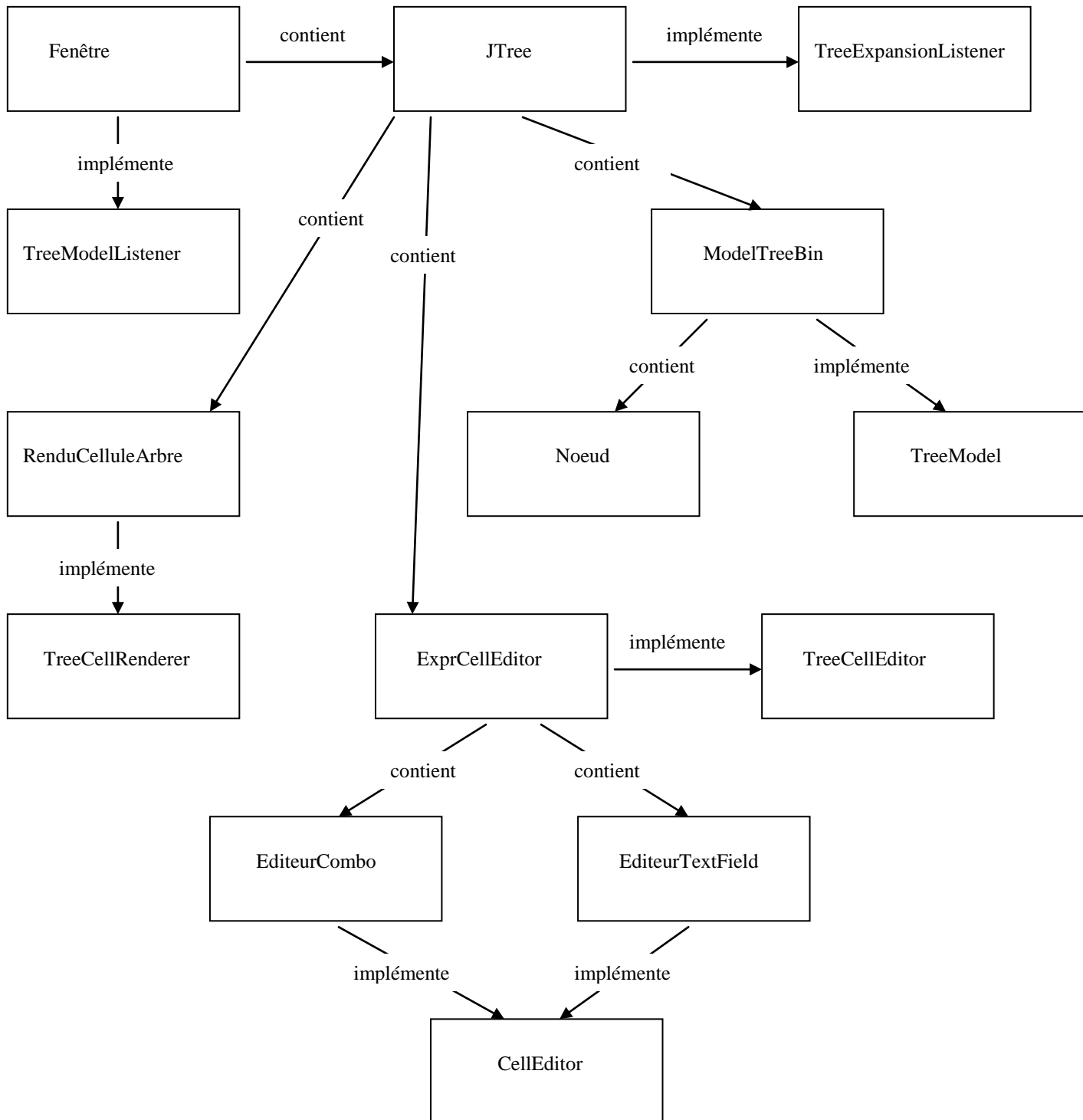
EditeurCombo (qui implémente CellEditor)

EditeurTextField (qui implémente CellEditor)

ExprCellEditor (qui implémente TreeCellEditor)

Exemple TreeBin8.

D'où le schéma des classes mises en jeu :



Travail à réaliser :

Vous allez tout d'abord utiliser un composant JTable pour afficher le résultat d'une requête sur la base de donnée (équivalent à l'usage d'un DataGrid en VB). Le composant JTable à un principe de fonctionnement qui suit la même logique que le JTree, le travail demandé demande d'utiliser le JTable dans un mode simple.

Ensuite, nous allons reprendre notre supervision des messages et alarmes, et passer l'interface en Swing. Ne vous laissez pas décourager par les étapes intermédiaires, ou les composants Swing se mélangent avec les composants AWT. Les comportements de l'interface sont quelques fois surprenants.

Le portage du composant camembert pose quelques problèmes dus aux changements de fonctionnement des composants Swing, particulièrement le panel. Pour redessiner, il ne faut pas définir `paint()` en Swing, mais `paintComponent()`, qui doit absolument appeler le comportement `paintComponent` de la classe mère.

Vous traiterez la liste des messages et des alarmes en rajoutant un icône dans la liste, permettant de distinguer les messages des alarmes, et les alarmes nouvelles, des alarmes prises en compte et des alarmes soldées.

Vous rajouterez la possibilité de choisir entre un camembert, et un histogramme pour chacun des deux graphes, et enfin, vous donnerez la possibilité de modifier les couleurs dans le camembert. Les couleurs seront sauvegardées, et réutilisées lors de la prochaine exécution de l'application.