ExpressO

Cours de programmation en langage JAVA pour étudiants connaissant le langage C

Patrick RUIZ 2019/2020


JAVA

Table des matières

1. LE LANGAGE 5

1.1.1. ORIENTE OBJET MAIS "FACILE" A MAITRISER 5
1.1.2. MULTITHREAD (LITTERALEMENT: MULTIFIL) 6
1.1.3. EDITION DE LIEN DYNAMIQUE 6
1.1.4. GESTION AUTOMATIQUE DES RESSOURCES MEMOIRE 6
1.1.5. MULTIPLATEFORME ET SUR 6
1.2. LA PLATE-FORME JAVA POUR LES ORDINATEURS 7
1.2.1. CONSTITUTION 7
1.2.2. COMMENT ÇA MARCHE? 8
1.3. NOUS VIVONS UNE EPOQUE FORMIDABLE (POUR L'INFORMATIQUE)! 9
1.4. LES "PACKAGES" DE L'API POUR ORDINATEUR (PAQUETAGES) 10
1.4.1. API (APPLICATION PROGRAMMING INTERFACE) JAVA « STANDARD » 11
1.4.2. UTILISER DES PAQUETAGES 11
1.4.3. CREER VOS PAQUETAGES 12

2. PROGRAMMATION «STRUCTUREE» 13

2.1. GENERALITES 13
2.1.1. VOCABULAIRE 13
2.1.2. ELEMENTS DE SYNTAXE 13
2.2. LES VARIABLES 14
2.2.1. NOTIONS COMMUNES 14
2.2.2. VARIABLES DE TYPES SIMPLE 14
2.2.3. VARIABLES DE TYPE REFERENCE (OBJET) 14
2.3. LES CONSTANTES 17
2.4. LES OPERATEURS 17
2.5. STRUCTURES DE CONTROLE DE FLUX 17
2.6. LES METHODES (EX FONCTIONS) 17

3. PROGRAMMATION ORIENTEE OBJET 18

3.1. CLASSES D'OBJETS 18
3.1.1. QU'EST CE QU'UNE CLASSE ? 18
3.1.2. QU'EST CE QU'UN OBJET ? 18
3.1.3. ET L'ENCAPSULATION ALORS ? 19
3.2. LES CLASSES EN DETAILS 20
3.2.1. LE CYCLE DE VIE D'UN OBJET 20
3.2.2. SYNTAXE DE LA DECLARATION D'UNE CLASSE PRINCIPALE 21
3.2.3. HERITAGE 22
3.2.4. SYNTAXE DE LA DECLARATION D'UN ATTRIBUT 24
3.2.5. SYNTAXE DE LA DECLARATION D'UNE METHODE 24
3.2.6. COMMUNICATION PAR MESSAGES (NECESSITANT UNE ASSOCIATION) 25
3.2.7. ATTRIBUTS ET METHODES DE CLASSE 26
3.2.8. SURCHARGE 26
3.2.9. SYNTAXE DE LA DECLARATION D'UN CONSTRUCTEUR 27
3.2.10. ENCAPSULATION : LE RETOUR 27
3.2.11. REDEFINITION DES METHODES ET POLYMORPHISME 28
3.3. ABSTRACTION DE LA COMMUNICATION PAR MESSAGES : INTERFACES. 28
3.3.1. CLASSES ABSTRAITES 28
3.3.2. INTERFACES 29

4. LES COMPOSANTS : JAVABEANS 30

4.1. QU'EST CE QU'UN COMPOSANT ? 30
4.1.1. DEFINITION: 30
4.1.2. INTERET: 30
4.1.3. JAVABEANS (GRAINS DE CAFE): 30
4.2. LES CLASSES INTERNES 31
4.2.1. DEFINITION 31
4.2.2. AUTRES PROPRIETES 31
4.2.3. PRINCIPALE UTILISATION : L'ADAPTATION 31
4.3. LES EVENEMENTS 34
4.3.1. PRINCIPE DE LA GESTION EVENEMENTIELLE 34
4.3.2. CLASSES ET INTERFACES LIEES A LA GESTION EVENEMENTIELLE 35
4.3.3. PLUSIEURS FAÇONS DE CODER 35

5. GESTION DES EXCEPTIONS 39

5.1. QU'EST CE QU'UNE EXCEPTION ? 39
5.2. LE MECANISME DE PROPAGATION DES EXCEPTIONS 39
5.3. ATTRAPER UNE EXCEPTION : LE BLOC `CATCH` , LE BLOC `FINALLY` 39
5.4. LANCER DES EXCEPTIONS : `THROW` ET `THROWS` 40

6. LES APPLETS 41

6.1. QU'EST CE QU'UNE APPLET ? 41
6.2. COMMENT LES APPLETS GERENT-ELLES LES ENTREES / SORTIES ? 42
6.3. LA QUESTION DE LA SECURITE 43
6.4. LA DIRECTIVE HTML <APPLET> 44

1. Le langage

Java *a été choisi par l'Open Handset Alliance*[i] *comme langage de développement pour le système d'exploitation Android (basé sur Linux). L'utilisation de Java avec le système d'exploitation Androïd est la principale application de Java pour l'informatique embarquée. Java dès son origine est un langage mature qui n'introduit pas de nouveaux concepts, mais intègre tous ceux qui ont fait leurs preuves dans les langages Orientés Objets qui l'ont précédé. Par exemple Java imite le C++ pour ses bons côtés et sa syntaxe, mais élimine ses défauts (lourdeur de la syntaxe pour la partie « orientée objet », manque de robustesse, code compilé obèse). Seule innovation propre à Java : il est un des seuls langages nés avec internet !*

1.1.1. Orienté objet mais "facile" à maîtriser

Java est **exclusivement** orienté objet : Il est donc conçu pour l'élaboration et la maintenance efficace des programmes les plus complexes et à la gestion des espaces mémoires les plus importants. Cependant contrairement à ses concurrents (C++, C#, Python), il ne possède pas une couche de programmation structurée pure. Donc, *en tant que langage à objets*, il est plus facile à maîtriser que ces derniers. En résumé, c'est le plus facile des langages orientés objet, mais malheureusement pour les débutants il n'efface pas les difficultés d'abstraction des langages à objets !

Intérêt des langages orientés objets:

1. **Structuration forte des données : Encapsulation**.
   Les données sont regroupées en structures cohérentes : Le modèle de structure est appelée une *classe* ; un ensemble de données organisées suivant le modèle fixé par la classe est un *objet* de cette classe. Les données sont encapsulées, dans le sens où seul le code écrit pour gérer la classe d'objets peut modifier les données. La notion de classe correspond donc à une structure de données, plus le code qui permet de gérer ces données.

2. **Recyclage simplifié du code : Héritage.**
   On peut créer une classe à partir d'une autre. La classe fille possède alors toutes les propriétés de la classe mère sans avoir à réécrire le code. Les classes constituent donc un ensemble hiérarchisé.

3. **Programmable grâce à des Ateliers de Génie Logiciel (Androïd Studio)**
   Si on veut parler de simplicité en terme d'apprentissage de la programmation, on peut dire que les ateliers de développement rapide (RAD) permettent une programmation très simple de Java. En effet dans ces environnements de développement, le programmeur se contente de créer des objets à partir de classes vendues par un développeur et d'établir des liens entre eux, la programmation s'effectue dans un atelier graphique.

[i] L'Open Handset Alliance (abrégé OHA, http://www.openhandsetalliance.com) est un consortium de plusieurs entreprises dont le but est de développer des normes ouvertes pour les appareils de téléphonie mobile. Créé en 2007 à l'initiative de Google il fédère 34 compagnies (Bouygues Telecom, China Telecom, Samsung, Motorola, Intel, Nvidia, eBay…). Il est a l'origine du développement du système d'exploitation Androïd.

1.1.2. Multithread (littéralement: multifil)

Java prévoit l'exécution concurrente de processus légers (programmation multitâche) par une programmation parfaitement cohérente avec le reste du langage. Ainsi *l'utilisation de toute la puissance des machines multicoeurs* est possible.

1.1.3. Edition de lien dynamique

Java présente des réponses au défit technologique qui consiste à *télécharger rapidement* du code, spécificité d'un langage destiné à internet:

1. **Appel du code contenu dans les bibliothèques *en cours d'exécution* du programme (cf §1.2.2.)** : Le code compilé Java, appelé *bytecode*, n'est pas exécutable, en effet *l'édition de lien est dynamique* ce qui permet de télécharger uniquement le code des classes nécessaires au fur et à mesure de l'exécution. Un grand nombre de bibliothèques se trouvent déjà dans machine : c'est le SDK pour Androïd (JRE pour la version destinée aux consoles) mis-à-jour régulièrement, elles n'ont donc pas à être téléchargées pour chaque application. Ainsi, la classe Java qui correspond au début de l'application ne fait en général que quelques kO (contrairement à un programme C++ exécutable par exemple) ce qui permet un téléchargement beaucoup plus rapide d'un programme écrit en Java.

2. **Arborescence des répertoires des bibliothèques (package) image de la hiérarchie de classes (cf §1.4.)**: Le nombre extraordinaire de classes mises à disposition par Androïd dans le SDK (https://developer.android.com/?hl=fr) pour les applications mobiles (ou dans le JDK par Oracle pour les ordinateurs, www.oracle.com) impose une organisation précise. **En Java cette organisation est une arborescence de répertoires**. Pour que l'édition de lien (link / éditeur de lien : linker) puisse se réaliser, *le code doit indiquer le chemin des classes* en local (sur le client) ou à distance (sur le serveur), c'est pourquoi **le nom des classes et des paquetages (groupes de classes) sont imposés par des règles strictes**.

1.1.4. Gestion automatique des ressources mémoire

La *simplicité* de Java en tant que langage à objets tient en partie à la gestion automatique des ressources mémoire , il n'est plus nécessaire de détruire les objets inutilisés: le garbage collector (ramasse miettes) supprime de la mémoire tout objet non utilisé (c'est-à-dire sans référence). Autre avantage, l'accés à la mémoire par pointeur est évité : ce qui rend plus difficile la création de virus , et simplifie notablement le code.

Par contre un inconvénient : la mise en route « automatique » du garbage collector *dessert* son déterminisme (qualité d'un code dont le temps d'exécution est *parfaitement* maîtrisé).

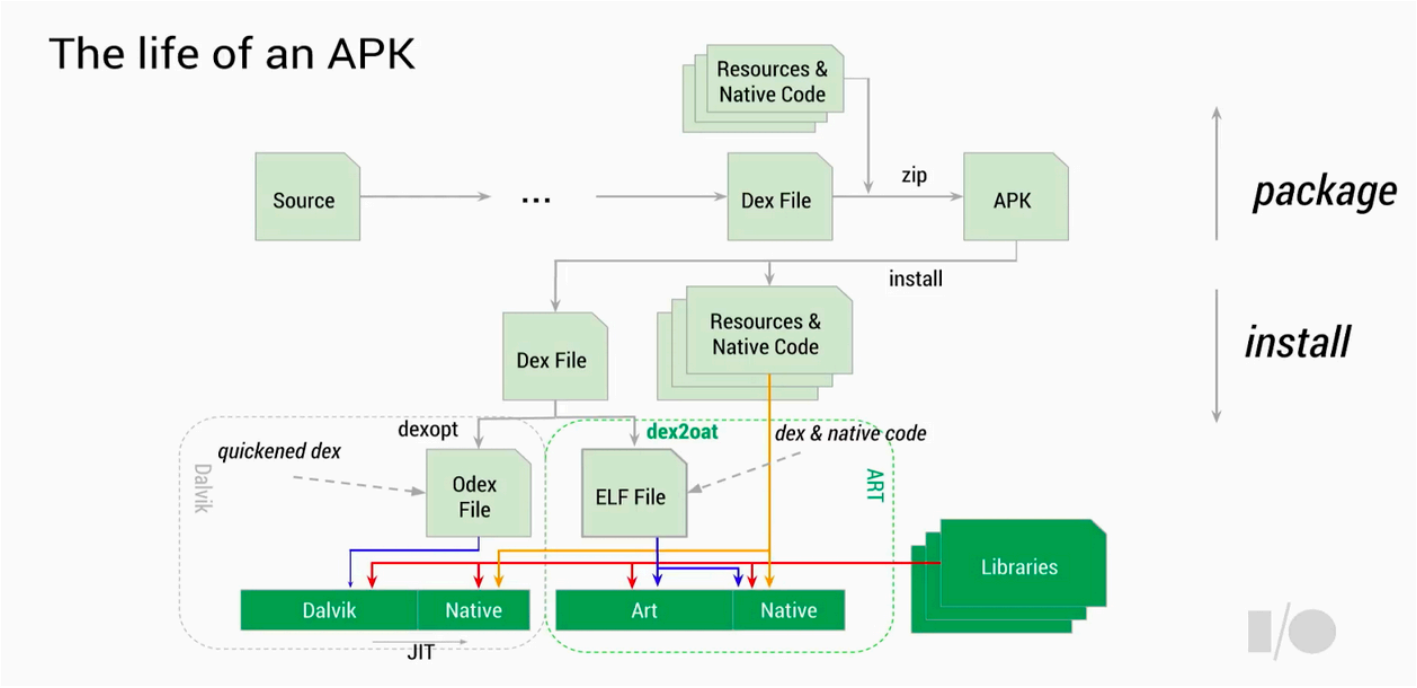
1.1.5. Multiplateforme et sûr

Pour assurer la portabilité du code Java sur toutes sortes de machines liées à l'internet Java utilise le concepts de machine virtuelle.

Pour les ordinateurs, la machine virtuelle se nomme JVM. C'est une couche logicielle entre le bytecode (code compilé Java) et le système d'exploitation de la machine (cf §1.2.1.). Cette couche adapte le code compilé Java à la machine utilisée, par ailleurs elle vérifie la non-dangerosité du code (recherche des virus).

Pour les machines sous système d'exploitation Androïd la machine virtuelle est remplacée par **l'Androïd Runtime (ART).** Le bycode associé à du « native code » son téléchargés. Après téléchargement *le code est compilé à l'avance* (AOT ahead-of-time) localement (dans le téléphone) en instructions natives ce qui produit un fichier ELF (Executable and Linkable Format). Ça pourrait

faire penser à un exécutable C normal (.exe), mais ce n'est pas le cas, puisque le fichier ELF n'intègre pas le code des bibliothèques du SDK: **l'édition de lien se fait toujours de façon dynamique** (à la manière des dll).



https://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/

1.2. La plate-forme Java pour les ordinateurs

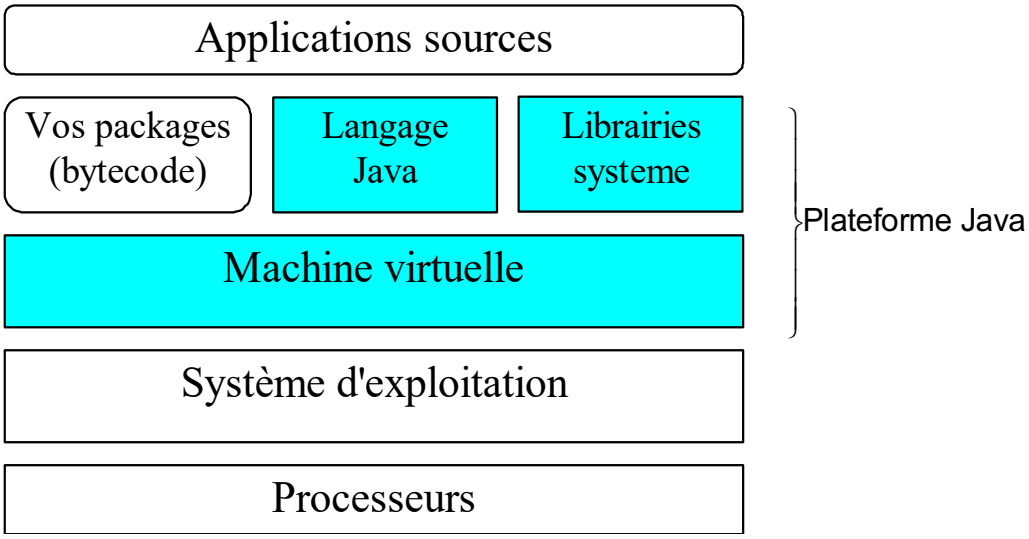
La plate-forme JRE (Java Runtime Environnement) est l'ensemble de la structure permettant le fonctionnement de programmes Java sur un ordinateur.

1.2.1. Constitution

1. Langage JAVA : normalisé par le Java Community Process (JCP).

2. Bibliothèque système : actuellement Java SE 8 (Java 2 Standart Edition 1.8) (cf §1.4.1).

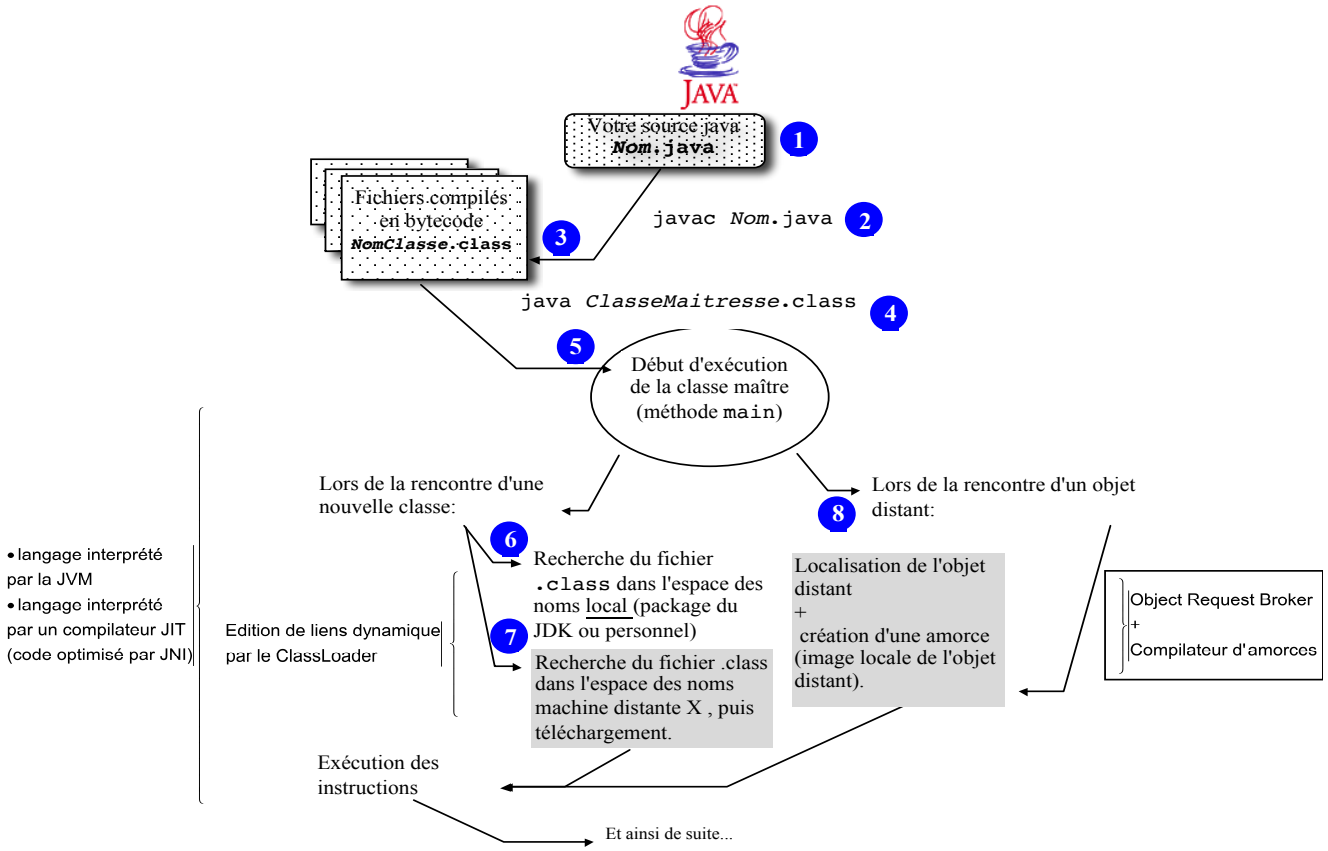
3. La machine virtuelle Java Virtual Machine JVM, qui est propre au système d'exploitation, permet au langage et à ses bibliothèques , elles universelles , de s'adapter à la spécificité de chaque machine , ce qui assure la PORTABILITE



1.2.2. Comment ça marche?

Commentaires du schéma page suivante :

1. Vous écrivez une classe *maîtresse* Java (ex: grâce à un éditeur de texte comme `notepad`) , c'est à dire *contenant une méthode main()* , et éventuellement d'autres classes dans un fichier portant le nom de la classe maîtresse et l'extension **`.java`**.
2. Vous la faites compiler en bytecode par le compilateur `javac` . Le bytecode est un langage compréhensible par la machine virtuelle, il ne s'agit pas du langage machine du processeur.
3. Résultat de cette compilation, un ou plusieurs fichiers. Chaque fichier correspond à une classe (principale) écrite dans le code source. Le nom de chaque fichier est constitué du nom de la classe, et de l'extension **`.class`**.
4. Pour lancer l'exécution il faut lancer la machine virtuelle en passant comme paramètre la classe maîtresse à exécuter. La machine virtuelle s'appelle `java`. Ex : java MaClasse
5. Le programme débute par la méthode `main` de cette classe.
6. A chaque fois qu'une classe nouvelle intervient dans le programme, l'édition de lien est effectuée par le Classloader (partie de la machine virtuelle). La classe est recherchée dans l'arborescence de répertoires correspondant aux packages (bibliothèques) grâce au chemin *inclus dans le code* (cf §1.4.).
7. Si la classe *maîtresse* a été téléchargée à partir d'un serveur, en cas d'échec lors de l'édition de liens, la classe est aussi recherchée sur le serveur.
8. Java permet aussi l'utilisation d'objets distants, c'est à dire d'objets créés sur le serveur. Leur utilisation ne sera pas étudiée dans le cadre de ce cours.



1.3. Nous vivons une époque formidable (pour l'informatique)!

1. **Java est mis à disposition gratuitement**.
   Téléchargement de l'environnement de développement : https://developer.android.com/studio
   Téléchargement adresse internet: **http://www.java.com**

2. **Java libère des applications et systèmes d'exploitation propriétaires** (windows, mac os, linux) : Vos programmes ne sont plus développés pour une seule plateforme, mais pour toutes à la fois. Ainsi, inutile d'écrire diverses versions.
   Par ailleurs, vous pouvez utilisez le système d'exploitation que vous préférez, car les applications Java ne lui sont pas dédiées. Donc, avec des applications Java, on n'est plus obligé d'utiliser tel ou tel système d'exploitation ! On peut prendre celui que l'on préfère !

3. Java offre une **norme universelle pour la programmation (lyrique)** : C'est le seul langage qui intègre parfaitement, car dès sa conception, les évolutions technologiques qui font la base de l'informatique du début du XXIe siècle (programmation objet, multitâche, multiplateforme, intégration aux navigateurs, utilisation d'objets distants, sécurité). Java est donc le langage qui s'adapte aux nécessités de tous les développeurs… bon il y a quand même un peu de place pour C++ et C quand même !

Année	Evénements
1995	mai : premier lancement commercial du JDK 1.0
1996	janvier : JDK 1.0.1
1996	septembre : lancement du JDC
1997	février : JDK 1.1
1998	décembre : lancement de J2SE 1.2 et du JCP
1999	décembre : lancement J2EE
2000	mai : J2SE 1.3
2002	février : J2SE 1.4
2004	septembre : J2SE 5.0
2006	mai : Java EE 5 décembre : Java SE 6.0
2008	décembre : Java FX 1.0
2009	février : JavaFX 1.1 juin : JavaFX 1.2 décembre : Java EE 6
2010	janvier : rachat de Sun par Oracle avril : JavaFX 1.3
2011	juillet : Java SE 7 octobre : JavaFX 2.0
2012	août : JavaFX 2.2
2013	juin : Java EE 7
2014	mars : Java SE 8, JavaFX 8

1.4. Les "packages" des API

Définition: Un paquetage ou "package" est un ensemble de classes et d'interfaces.
Le paquetage est donc un concept semblable aux bibliothèques du C.

Intérêt:

- Permet de retrouver plus facilement une classe.
- Evite les conflits de nom (en effet deux classes de deux package différents peuvent avoir le même nom sous certaines conditions).
- Définir un niveau supplémentaire d'encapsulation

1.4.1. API (Application Programming Interface) Java « standard »

Les API Java sont en fait des paquetages fournis par Oracle ou Androïd avec le langage, qui regroupent des classes par domaines d'utilisation.

La documentation des packages Androïd : https://developer.android.com/reference/packages

La documentation des packages Oracle (pour ordinateurs) :
https://docs.oracle.com/javase/8/docs/api/

Voici les API communes commencent toutes par « java.» . Voici celles de base :

- API langage : le paquetage **java.lang** , fournit les classes indispensables au langage:

la classe *Object* (mère des classes) , chaînes de caractères, nombres (entier, à virgule ..) , interactions avec le systeme (JVM) , fonctions mathématiques , gestion des threads , gestion des exceptions.

- API utilitaires : le paquetage **java.util** , fournit

- les classes permettant la création de structures de données : hashtable, vecteur, piles, énumérations.
- les classes permettant la gestion des dates (calendrier).
- les classes permettant la gestion des évènements et des modifications d'objets.

- API gestion des entrées/sorties : le paquetage **java.io** , est constitué des classes gérant les flux d'octets (buffers...), de caractères , les accès aux fichiers.

Il y en a des centaines d'autres communes ou différentes…

1.4.2. Utiliser des paquetages

Pour programmer en Java vous êtes obligés d'utilisez un minimum de classes fournies les API. Comment s'y prendre ?

Les classes d'un paquetage ne sont accessibles que par les classes du même paquetage à moins qu'elles ne soient déclarées `public`.

Pour utiliser la classe d'un paquetage on peut l'appeler par son nom complet :
Syntaxe: *`nomPaquetage.NomClasse`*
Ex: pour la classe `Button` qui représente un bouton il faut écrire: `java.swing.Button`.

Pour pouvoir utiliser son nom court *`NomClasse`* , il faut importer la classe : `import`
Syntaxe:
`import nomPaquetage.NomClasse ;`
Ex: pour la classe `Button` qui représente un bouton il faut écrire: `import java.swing.Button`.

On peut aussi importer tout un paquetage (toutes les classes d'un coup) :
Syntaxe: `import nomPaquetage.* ;`
Ex: pour importer tous les éléments d'une GUI[ii] : `import java.swing.*`;

En cas d'ambiguïté de nom de classe une erreur est levée.

Trois paquetages sont **importés automatiquement**:

- `java.lang`

[ii] Graphic User Interface

- le paquetage par défaut (paquetage sans nom, à la racine des paquetages).
- le paquetage courant (toutes les classes du répertoire où est située la classe)

1.4.3. Créer vos paquetages

A partir du moment ou vous programmez en Java, vous écrivez vos propres classes. Celles-ci font forcément partie d'un paquetage. Lequel ?

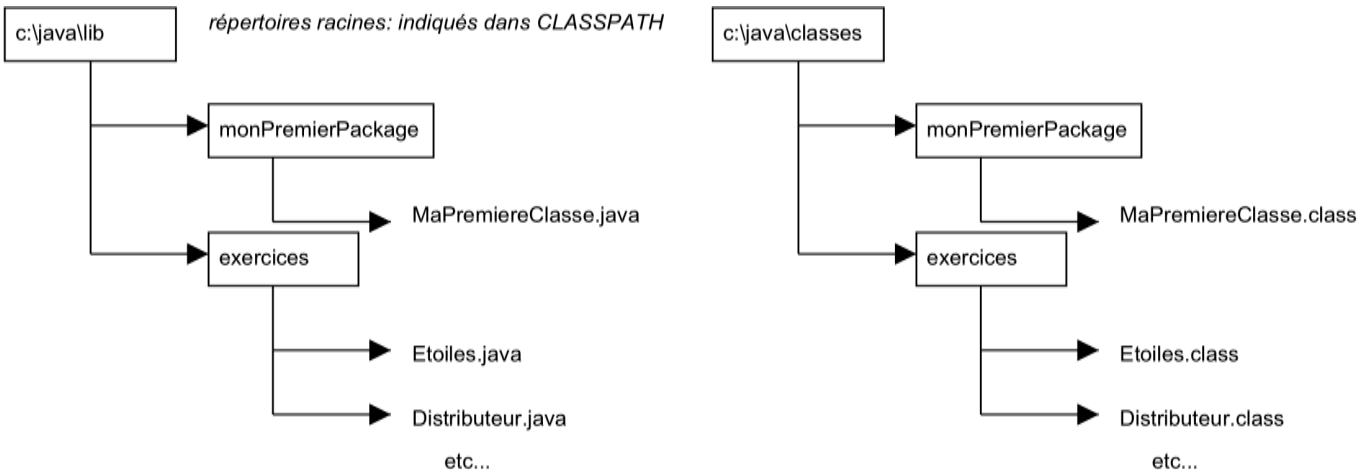
Le JDK impose une structure de fichiers précise pour les paquetages:

- Le nom du fichier contenant une classe publique doit être identique à celui de la classe suivi de `.java`. Ex: `MaPremiereClasse` est enregistrée dans `MaPremiereClasse.java`
- Les classes d'un paquetage doivent être placées dans un répertoire *dont le nom est celui du paquetage*. Ex: `MaPremiereClasse` fait partie du répertoire `monPremierPackage`.
  Ex: Mise en parallèle du chemin `monPremierPackage\MaPremiereClasse.java`
  et du nom de la classe `monPremierPackage.MaPremiereClasse`
- La variable d'environnement `CLASSPATH` contient le (ou les) répertoire racine des paquetages. Par défaut : le répertoire courant est c:\...\lib\classes.zip

Le code source de la classe doit aussi contenir le nom de son paquetage : `package`
Syntaxe: `package` *`nomPaquetage`*`;`
En cas d'oubli votre classe se retrouve dans le paquetage par défaut.

Qu'en est-il des classes pré-compilées ?

Elles doivent être situées dans une hiérarchie de répertoires identique à celle décrite plus haut, soit la même soit débutant sur un autre répertoire racine (ce qui permet de protéger votre code source). Bien entendu il faut que `CLASSPATH` contienne le chemin de ce répertoire.



Remarques:

1. Certains environnements de développement peuvent utiliser leur propre variable d'environnement, mais ils tiennent tous compte de CLASSPATH.
2. Pour se simplifier la vie il vaut mieux au début ne pas chercher à déplacer les packages.

2. Programmation «structurée»

Un langage de programmation objet peut être vu comme étant constitué d'un langage de programmation structurée auquel s'ajoutent la syntaxe permettant de gérer des classes d'objets. Puisque ce cours s'adresse à des étudiants qui possèdent la programmation structurée en langage C, nous commençons par ce chapitre pour montrer le parallèle C/Java. Mais il faut toujours garder à l'esprit que la programmation structurée en Java ne peut se pratiquer qu'au sein d'une classe !

2.1. Généralités

2.1.1. Vocabulaire

La programmation objet suppose l'utilisation d'objets et de classes d'objets :

- **Objet (ou instance):** Groupe de variables (semblable à une variable de type structure en C) pour lesquelles on possède aussi un groupe de fonctions pour les manipuler. Ceci forme un tout indissociable.
- **Classe d'objets**: Description servant de **modèle** pour la construction d'objets de cette classe. Il s'agit d'une **description** de l'organisation des variables qui seront déclarées automatiquement à la déclaration d'un objet de la classe, plus une description de toutes les fonctions permettant les manipulations des données des objets.

Par rapport au langage C, deux termes nouveaux:

- **Attribut (ou propriété):** une variable *membre* d'un objet (attribut d'objet) ou d'une classe (attribut de classe).
- **Méthode (ou message):** une fonction (d'objet ou de classe).

Et un terme à usage unique :

- **Variable :** variable locale (sa portée est le *bloc* où elle est crée, pour mémoire un bloc est compris entre { } comme en langage C).

2.1.2. Eléments de syntaxe

- Le programme commence par la méthode `main` d'une classe `public`.
  Cette méthode `main` , bien qu'écrite au sein d'une classe existe indépendamment de tout objet de la classe (mot clef `static`). Il faut la percevoir comme étant à l'extérieur de toute classe, elle constitue **une amorce du programme alors qu'aucun objet n'existe encore**.

  Cependant, en Java , tout code doit être inclus dans une classe (philosophie du langage) , c'est pourquoi la méthode `main` se trouve au sein d'une classe, cette classe est une classe *maîtresse* , c'est à dire une classe à partir de laquelle on peut lancer l'exécution grâce à la machine virtuelle (par la commande `java Exemple.class`).

```
package exercices;
```

```
public class Exemple {
     public static void main (String args[]) {
     ...
     }
}
```

Comme en C:

- Les instructions sont *séparées* par des **;**
- Les *blocs* d'instructions sont compris entre {...}.
- Les *commentaires* : /*..*/ ou //
- Java différentie les majuscules des minuscules.

A la différence du C:

- Il n'existe *pas* de préprocesseur (`#include` , `#define` etc...).

2.2. Les variables

2.2.1. Notions communes

Une variable n'est pas un attribut !

Variables **locales de méthodes** : Utilisation identique aux variables locales du C. Leur portée est celle de la méthode (ou du bloc) où elles sont déclarées. (il existe aussi des variables **locales** de classe : voir plus loin)

Variables **globales** : Elles n'existent pas en Java !

Un conseil : Le **nom** faites commencer par une **m**inuscule, c'est une règle de style toujours appliquée en Java.

Les variables peuvent être de type **simple** (int, float…) ou de type **référence** (c'est l »équivalent du «pointeur» en C).

2.2.2. Variables de types simple[iii]

Types simples : ***`byte` (1 octet!) , `boolean` (1bit=`true` / `false`)*** , `short` (2 octets) , `int` (4 octets) , `long` (8 octets) , `float` (4 octets) , `double` (8 octets) , `char` (2 octets = Unicode).

Les **conversions** se font implicitement d'un type faible à un type fort (comme en C) mais déclenchent une erreur de compilation dans le sens contraire (à la différence du C).

Initialisation impérative avant utilisation (sous peine d'erreur).

Manipulation par valeur obligatoire : contrairement au langage C **on ne peut accéder au pointeur de ces variables**.

2.2.3. Variables de type référence[iv] (objet)

[iii] Parfois appelées PRIMITIVES
[iv] Parfois appelées HANDLE

Une variable de cette catégorie contient **la référence *à un objet*** (jamais à une autre variable, uniquement un objet). C'est à dire qu'elle pointe en mémoire à l'endroit où est placé l'objet, elle correspond à l'adresse de l'objet (via la machine virtuelle… ce n'est donc pas l'adresse physique, et de toutes façons sa valeur n'est pas lisible).

On peut rapprocher cette catégorie de variables aux variables de type structure en C[v].

Certains **objets** *référencés* (on n'utilise pas le terme pointé) par des variables de type référence sont d'usage très courants : les tableaux et les chaînes de caractères par exemple. Ils bénéficient de ce fait d'une syntaxe spécifique. Attention il s'agit bien d'objets en Java ! Ce qui signifie qu'ils disposent de méthodes pour les manipuler ! Nous détaillons donc ici leur utilisation très particulière.

a. Tableaux

Ceux sont des objets, pour les utiliser il faut:

- Déclarer la référence à un tableau, nom de la variable qui le pointe: `type[] nom ;`
- Le créer en mémoire: `nom = new type[dim1][dim2];`
- Le définir : `nom[indice]= valeur;`
  L'indice est forcément un `int`.

Comme en C on peut tout faire d'un coup: `type[] nom = new type {val1,val2}`
Autre version : `type nom []= new type {val1, val2}`
Comme en C les indices vont de 0 à la dimension du tableau -1.

Contrairement au C :

- tout dépassement déclenche une erreur
- la dimension du tableau est fournie par la méthode `lenght ()`, car les tableaux sont en fait des objets de la classe des tableaux d'un type donné (vous suivez ?).

Le type peut être un type simple ou composé (des objets par exemple).

b. Chaînes de caractères

Ce sont aussi des objets ! Deux classes sont disponibles: `String` et `StringBuffer`.

La classe `String` défini des chaînes de caractères **constantes**. Elle est de loin la plus utilisée (notamment pour les passages de paramètres aux méthodes).
Ce sont des objets, pour les utiliser il faut faire comme avec tous les objets:
*
Déclarer le nom de la chaîne de caractères :`String nom ;`
*
L'instancier (la créer en mémoire) et la définir (lui donner une valeur):

```
nom = new String (chaîne_de_caractères);
ou tout simplement
nom = chaîne_de_caractères;
```

*
Mais on peut aussi définir des chaînes de caractères constantes sans nom. Pour ces chaînes de caractères la création est automatique : `"bonjour"`

La classe `StringBuffer` défini des chaînes de caractères de contenu **variable**.

[v] On sait qu'en C le nom d'une structure est en fait un pointeur sur la variable de type structure. En Java on peut dire que les pointeurs n'existent pas au sens du C , car on ne peut pas accéder à l'adresse de l'objet grâce à une variable référence , ni faire des opérations sur ces variables comme on le fait avec les pointeurs.

Ceux sont des objets , pour les utiliser il faut:
*
Déclarer le nom de la chaîne de caractères :`StringBuffer nom ;`
*
Le créer en mémoire : `nom = new StringBuffer (longeur);`
On est autorisé à ne pas préciser la longueur[vi] , elle est gérée automatiquement ! Cependant un programme sera ralenti par cette gestion dynamique , il vaut mieux prévoir une longueur adaptée.
*
Le définir : `nom = chaîne_de_caractères;`

Un luxe : La concaténation grâce à l'opérateur + !
C'est la méthode utilisée dans certains exemples :
```
System.out.println("bonjour : " + prenom );
```
*
Syntaxe: `string_ou_var_prim + string_ou_var_prim`
*
Le résultat est un nouvel objet `string`.
Les conversions se font automatiquement par appel des méthodes `toString()`.

Les conversions.
1. La méthode `valueOf` se charge de toutes les conversions pour les types simples:
*
Variables primitives vers `string` : `string.valueOf(var)` renvoi un `string`.
*
`string` vers variables primitives : `type.valueOf(chaîne)` renvoi un résultat du `type` précisé.
2. La méthode `toString` se charge des conversions pour les types références:
On écrit en général dans toutes les classes une méthode `toString` qui permet de fournir pour les objets une chaîne de caractère qui renseigne sur l'état de l'objet.

Comment modifier les `StringBuffer`?
Grâce aux méthodes fournies par cette classe:
*
La plus courante est `append` qui rajoute à la suite les caractères entre parenthèse. Cette méthode effectue les conversions automatiquement (comme pour la concaténation).
Exemple:
```
StringBuffer test=new StringBuffer();
test.append(true);
test.append(" ");
test.append('C');
test.append(2);
```
*
II existe aussi `insert` qui comme on peut s'en douter à insère une sous-chaîne.
*
`setCharAt` fait la même chose mais pour un caractère isolé.

Passage d'arguments via la ligne de commande.
Le tableau `String args []` contient les paramètres de la ligne de commande indéxés comme tous les tableaux à partir de 0. On rappelle que chaque paramètre différent doit être séparé par un espace.

[vi] La longueur par défaut est 16 , c'est la capacité de la chaîne (cf p 62)

c. Les structures et les unions

Ces structures de données n'existent plus en Java. Les unions ne présentaient pas grand intérêt. Quand aux structures, on a déjà dit qu'elles sont *une forme primitive d'objet*, en effet on peut construire une structure si on construit une classe d'objets ne présentant aucune méthode et dont tous les attributs sont publics.

2.3. Les constantes

Syntaxe de déclaration : `final type nom ;`

L'initialisation peut être faite séparément de la déclaration.

Attention : Ces constantes sont locales.

2.4. Les opérateurs

Semblables à ceux utilisés en C[vii].

2.5. Structures de contrôle de flux

On retrouve tous les contrôles du C.

Une nuance : l'expression logique donne un résultat de type `boolean`, donc un opérateur donnant un résultat non booléen ne peut être utilisé comme expression logique (contrairement au C).

2.6. Les méthodes (ex fonctions)

Elles doivent apparaître *systématiquement* au sein d'une classe.

Mais il n'y a plus à donner de prototype !

[vii] L'opérateur ternaire existe encore !

3. Programmation orientée objet

3.1. Classes d'objets

3.1.1. Qu'est ce qu'une classe ?

C'est un **modèle** de structure de données composé de membres de deux natures:

*
Les attributs : Structure des données.
*
Les méthodes : Actions qui peuvent être effectuées sur les données.
Une de ces méthodes peut être `main` , elle constitue alors le début du programme cf § 2.1.2.

En plus de ces membres , la classe contient un élément d'une autre nature :

- Le constructeur : Les constructeurs ressemblent à des méthodes , mais ils ne sont utilisés qu'à la création des objets de la classe. Ils servent à l' **initialisation** de l'objet.

Exemple:

```
public class Point {

// Attributs
  private double x;                 // x abscisse
  private double y;                 // y ordonnée

// Constructeur
Point (double u, double v) {
    x=u;
    y=v;
  }

// Méthodes

public double getX () {             // Accesseur abscisse
    return x;
  }

public double getY () {             // Accesseur ordonnée
    return y;
  }


}
```

3.1.2. Qu'est ce qu'un objet ?

C'est un ensemble de données structurées suivant le modèle que fournit sa classe.

Voc: Instancier : Réserver de la mémoire pour un Objet Instance : Autre terme pour objet.

Exemple:

```
import java.io.*

class ExemplePoint {

  public static void main (String args[]) {
    Point m, n, w, z;       // Déclaration des objets Point

    m = new Point(1,2);     // Création de l'objet m , initialisation
    System.out.println("M: " + m.getX() + " + " ; "+ m.getY());
    n = new Point(3,-4.5);
    System.out.println("N: " + n.getX() + " ; " + n.getY());
  }
}
```

La classe `ExemplePoint` n'a pour intérêt que de fournir une méthode `main` , en effet il ne peut exister de code en dehors d'une classe[viii].

Dans `main` on vient créer des objets de la classe Point.
Comment s'y prend-on ?

Grâce à new : mot clef qui

1. réserve l'espace mémoire nécessaire pour l'objet (instantiation) puis
2. appelle le constructeur (définition ou initialisation de l'objet) ici `Point(..)`

Réflexion: w et z existent-ils en mémoire ? Que signifie leur déclaration ?[ix]

3.1.3. Et l'encapsulation alors ?

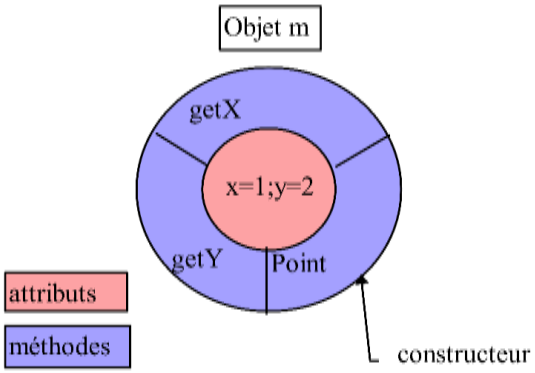
C'est le concept qui évite :

1. **La corruption des données** contenues dans l'objet par du code qui malencontreusement emploierait ces données alors qu'il ne le devrait pas (erreur de programmation courante lorsque plusieurs développeurs travaillent sur le même projet).
2. La **nécessité de connaître l'implémentation** des méthodes de l'objet lorsqu'on l'utilise. En effet les méthodes suffisent à contrôler les données de l'objet , il est donc inutile de savoir comment est programmé en détail l'objet. Bien entendu , pour avoir assez de souplesse , il faut que les méthodes soient bien conçues.

Par encapsulation on entend l'impossibilité pour un objet d'une classe d'accéder directement (sans passer par des méthodes) à un attribut appartenant à un objet d'une autre classe.

Ce concept est représenté par le schéma suivant:

[viii] Puisque `main` est `static` , elle peut être appelée sans avoir a instancier `ExemplePoint` (cf §3.2.7). C'est la machine virtuelle qui l'appelera à l' exécution de `java ExemplePoint` à partir de la fenêtre DOS.
[ix] w et z sont des variables références : Ces variables peuvent pointer sur des objets de la classe Complexe , mais pour l'instant elle ne se voient affecter la référence à aucun objet , elle contiennent donc la valeur NULL. Tandis que m et n contiennent l'adresse des objets Point (1,2) et Point (3,-4.5). Elles pointent sur ces objets. On pourrait modifier l'objet pointé , par exemple modifiez le programme pour que w et m pointent à la fois sur le même objet.



Les attributs ne peuvent être atteints sans passer par une des méthodes , ou le constructeur.

Il existe 4 niveaux d'encapsulation pour les attributs et les méthodes , les deux niveaux les plus utilisés étant :

- `protected` : Accès autorisé aux objets de la classe , des classes filles et des classes du même paquetage.
  Utilisation : Pour les éléments non publics a transmettre par héritage.

- `public` : Accès autorisé à tous les objets.
  Utilisation : Niveau des membres permettant la communication entre les objets.
  A éviter pour les attributs. Conseillé pour les méthodes.

Dans le schéma, les attributs sont `protected` et les méthodes `public`.

A ne jamais faire:
Court-circuiter le concept d'encapsulation en rendant publiques les attributs. Dans ce cas on rend l'objet vulnérable... On perd donc le bénéfice de la programmation objet.

3.2. Les classes en détails

3.2.1. Le cycle de vie d'un objet

1. **Création de l'objet en trois étapes**

   - **Déclaration** : Création d'une variable portant le nom indiqué et contenant une référence (adresse) à un objet du type indiqué (une classe). Syntaxe : ***`type nom ;`***
   `nom` est une variable référence (une étiquette) pour la classe *`type`*.

   - **Instantiation** : Création de l'espace mémoire nécessaire à l'utilisation de l'objet. Syntaxe: *`nom`* `= new` *`type`*`();`
`nom` contient l'adresse de l'espace mémoire[x] qui est réservé un objet de la classe *`type`*

- **Initialisation** : Affectation de valeurs aux attributs

[x] Avant cette instruction , *`nom`* contenait `NULL` c'est-à-dire le pointeur nul , aucune adresse !

. par un **constructeur** de l'objet[xi].
Syntaxe: *nom* = new *type (données);*
. par défaut les valeurs sont égales à 0.
. par un **initialiseur** (bloc d'initialisation par défaut). PEU UTILISE.

EXEMPLE: `Hexagone hexa = new Hexagone(50, new Point(100,100));`
Création d'un objet Cercle de rayon 50 et de centre l'objet de type `Point` qui est un point en {100;100}

2. Utilisation de l'objet:

- Grâce aux **méthodes** :
Syntaxe: *nom_objet.nom_méthode(paramètres).*
EXEMPLE: `cercle1.tracer();`

- Grâce aux **attributs** : **A éviter en POO !** Java permet de restreindre l'accès aux attributs en les protégeant par `protected` : cf. Encapsulation.
Pour utiliser l'attribut d'un objet il faut utiliser la syntaxe: *nom_objet.nom_attribut*[xii].

Réflexion : Quel risque prend-t-on lorsqu'on autorise l'accès à un attribut ?[xiii]

3. Destruction de l'objet:

- Elle est gérée **automatiquement** par le ramasse miettes (garbage collector)! Celui-ci efface régulièrement tous les objets qui n'ont pas de variable référence (références hors de portée , inexistante ou mise à `null`).
- Avant la destruction de l'objet vous pouvez désirer effectuer certaines tâches ménagères comme libérer les références à d'autres objets. Ceci est possible grâce à la méthode `finalize()`(luxe)
.

3.2.2. Syntaxe de la déclaration d'une classe principale

Ceux sont les classes les plus souvent utilisées. Leur nom correspond au fichier `.class` généré lors de la compilation.

Syntaxe:
Tout ce qui est entre [ ] est optionnel.

[*Modificateur1 Modificateur2*] **`class`** ***`Nom_classe`*** [`extends` *Nom_classe_mère*] [`implements` *Nom_interface1, Nom_interface1*] **{**

```
// corps de la classe , ordre habituel des membres de la classe

  attributs

  [
static { ... // initialiseur de classe
}

{...// initialiseur d'instance
}
]
```

[xi] Le mécanisme est identique au passage de valeurs par paramètre à une fonction.
[xii] Ceci correspond exactement à l'utilisation des membres d'une structure !
[xiii] Que l'indice soit modifié au delà des limites du tableau contenant les points!

constructeurs

méthodes publiques

methodes privées

}

Le **nom** de la classe commence par une **M**ajuscule !

- Il existe 3 modificateurs :
  - `public` : seule classe accessible en dehors du paquetage. Les classes non publiques ne peuvent être utilisées que par les classes du même paquetage.
  - `abstract` : classe abstraite , on ne peut l'implémentée. Voir plus loin.
  - `final` : classe de fin de hiérarchie , on ne peut en fabriquer des sous-classes.

  On peut en utiliser plusieurs à la fois.

Par défaut la classe est non publique , non abstraite et non finale.

- `extends` indique que la classe hérite (ou étend) de la classe mère *Nom_classe_mère*. Par défaut la classe hérite de la classe `Object`.
- `implements` indique que la classe implémente les interfaces *Nom_interface1, Nom_interface1*. Par défaut une classe n'implémente aucune interface.

Les initialiseurs : Blocs de code lu juste après la création de la classe ou d'une instance et qui est utilisé en général à l'initialisation des attributs. Il peut y en avoir autant qu'on veut.

3.2.3. Héritage

C'est le concept qui permet une récupération facile du code déjà écrit.

Lorsqu'une sous-classe (classe fille) hérite d'une super-classe (classe mère) elle possède d'emblée tous les membres de sa super-classe (sauf ceux qui ne lui sont pas accessibles par encapsulation).

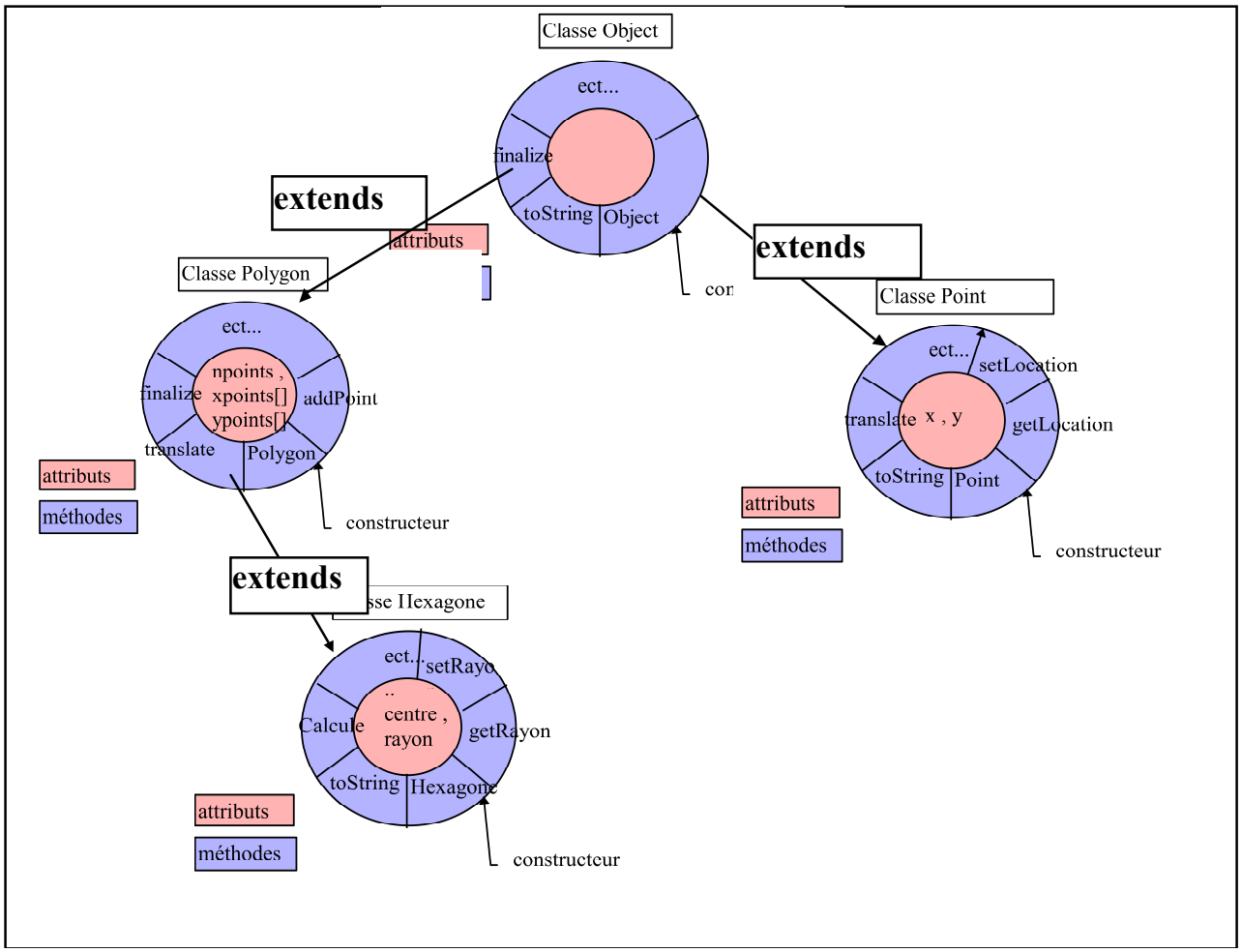
Remarques:
- Toute modification apportée à la classe mère se répercute sur la classe fille.
- L'héritage est une propriété des classes pas des objets !
- Les constructeurs ne sont pas hérités : ceux ne sont pas des membres de la classe à part entière.
- On ne peut étendre une classe déclarée `final`.

Exemple: Supposons que l'on désire créer un héxagone, on peut alors récupérer le travail effectué pour les polygone par les créateurs de Java en étendant la classe `Polygon` dans la classe `Hexagone`.

```
package exercices;

import java.awt.*;
```

```
public class Hexagone extends Polygon {

  int rayon=10;// Attributs supplémentaires
  Point centre=new Point (0,0);

  public Hexagone(int leRayon , Point leCentre){//Constructeur
  rayon=leRayon;
  centre=leCentre;
  }

  private void calcule() {// Méthode privée supplémentaire
  this.addPoint(centre.x,centre.y+rayon);
  this.addPoint(centre.x-(int)(rayon*0.866),centre.y+rayon/2);
  this.addPoint(centre.x-(int)(rayon*0.866),centre.y-rayon/2);
  this.addPoint(centre.x,centre.y-rayon);
  this.addPoint(centre.x+(int)(rayon*0.866),centre.y-rayon/2);
  this.addPoint(centre.x+(int)(rayon*0.866),centre.y+rayon/2);
  }

  //   etc... , on peut rajouter getRayon ,...

}
```

Ce qui nous donne la hiérarchie de classes suivante[xiv]:



[xiv] La classe Point ne correspond pas à celle qui a servi d'exemple , mais à la classe fournie par SUN dans java.awt

3.2.4. Syntaxe de la déclaration d'un attribut

Tout ce qui est entre [ ] est optionnel.
Syntaxe: [*Modificateur1 Modificateur2 ...*] ***type nom;***

EXEMPLE : `private double x;  // x abscisse`

Différence avec les variables : Déclaration dans le corps d'une classe (en dehors de tout bloc).

Le nom commence par une **m**inuscule.

Les modificateurs peuvent être:

- Les niveaux d'accès (encapsulation) : `private` , `protected` , `public`. Le niveau **par défaut** est appelé niveau paquetage (sans mot clé).
- `final` : Une constante.
- `static` : déclare qu'il s'agit d'un attribut de classe.

3.2.5. Syntaxe de la déclaration d'une méthode

Tout ce qui est entre [ ] est optionnel.
Syntaxe:
[*Modificateur1 Modificateur2 ...*] ***type_valeur_retournée nom (paramètres)*** [`throws` ***exceptions***]

{
corps de la méthode
}

Le **nom** commence par une **m**inuscule.

Comme en C, la valeur retournée est celle précisée dans l'instruction `return` trouvée dans la méthode. Mais en Java on peut souvent s'en passer.

Remarque : Le type de la valeur retournée doit coïncider avec la valeur contenue dans return sauf pour un objet ou **le type peut aussi être une sous-classe** de la classe de l'objet. Ex: On pourrait déclarer dans la méthode `Hexagone`

```
Dilate (Polygon Hexa){

...

return (Hexa)}
```

- `throws` : spécification des exceptions levées par la méthode (voir plus loin).

Les modificateurs peuvent être:

- Niveaux d'accès (encapsulation) : `private` , `protected` , `public`. Le niveau par défaut est appelé niveau paquetage (sans mot clé).
- `final` : Une méthode fixée définitivement.
- `static` : déclare qu'il s'agit d'une méthode de classe.

- `abstract` : Méthode non implémentée.

- `native` : Méthode écrite dans un autre langage (le C par exemple!) .

- `synchronised` : Méthode accédant simultanément à une autre méthode aux données (multithreads).

3.2.6. Communication par messages (nécessitant une association)

C'est le concept qui permet l'échange d'informations entre les différents objets qui constituent le programme sans casser l'encapsulation , donc grâce aux méthodes.

Comment agir sur un objet à partir de lui-même ?

Les méthodes de l'objet peuvent utiliser tous les attributs de l'objet comme s'il s'agissait de variables locales à la méthode.

Comment agir sur un objet2 à partir d'un objet1 ?

- A partir d'un objet de la même classe : Il y a accès libre aux attributs.
- A partir d'un objet d'une classe différente :
  - Invocation des méthodes de l'objet2 (lorsque l'accès est autorisé).
  - Accès direct aux attributs lorsque l'accès est autorisé (à éviter en POO).

Transmettre des informations d'un objet1 à un objet2 d'une classe différente:
Passage de paramètres lors de l'invocation d'une méthode de l'objet2.

Le passage de paramètres se fait par valeur, donc :
- Une méthode ne peut modifier la valeur d'une **variable de type primitif** passée en paramètre (car elle est en dehors de sa portée).

- Puisque la valeur contenue dans le nom d'une **variable objet** est l'adresse de l'objet , une méthode de l'objet2 peut modifier l'objet1 dans la mesure où l'accès est autorisé (en fait c'est un passage par adresse sans le dire).

Remarque: Les **noms des paramètres** d'une méthode peuvent être les **même que ceux des attributs** , mais alors ces noms **cachent les attributs** de l'objet, il faut utiliser `this.nomAttribut`.

Recupérer les informations d'un objet2 dans un objet1 de classe différente:

- Les **accesseurs** : Ceux sont des méthodes écrites pour informer sur les *attributs importants* de l'objet (qu'on appelle les ***propriétés***). Il est conseillé de les nommer `getNom_attribut.`
- Valeur de retour d'une méthode de l'objet2 (peu utilisé). Cette valeur peut être un objet, auquel cas on transmet l'adresse de l'objet, on va donc pouvoir agir sur cet objet.
  Mot clé: `this` permet de renvoyer l'adresse de l'objet courant , objet2.
- Accès direct aux attributs lorsque l'accès est autorisé (normalement interdit en POO).

3.2.7. Attributs et méthodes de classe

Jusqu'à présent nous n'avons étudié que des attributs et des méthodes **d'instance** :
Les attributs appartiennent à un objet et les méthodes agissent sur un objet.

Les attributs et méthodes **de classe** sont déclarés grâce au modificateur `static`.

Tous les objets d'une classe partagent les **attributs de classe** : il en existe qu'un de chaque en mémoire , auquel peuvent accèder tous les objets de la classe. C'est ainsi qu'on définit les constantes par exemple.

Les **méthodes de classe** ne peuvent agir que sur des attributs de classe (attributs d' objet interdits). Ce type de méthodes est très rare en POO. Un exemple est `print`.

On peut accéder directement aux attributs et méthodes de classe à partir du nom de la classe : il est inutile de créer une instance de la classe pour y accéder.

Syntaxe : *`NomClasse.nomMembreStatic`*.

Comment initialiser les attributs de classe ?

- Dans sa déclaration (aussi possible pour les attributs d'instance) , syntaxe :
  `class` *`NomClasse`* `{`
  `static` *`type nomAttribut`*`=10;`
  `etc ...`
  Surtout utilisé pour les attributs de type primitif.

- Par un initialiseur de classe , exemple :

```
class Salaires{
static String [] employé;
static {

                                                                    /
/ initialiseur de classe
employé[0]="Patron";
employé[1]="Secrétaire";
employé[2]="agent comptable";
}
etc ...
```

  Surtout utilisé pour les attributs de classe de type objet.
  Un initialiseur de classe n'est effectué qu'une fois , à la création de la classe.

3.2.8. Surcharge

Plusieurs méthodes d'une même classe peuvent posséder le même nom ...
... mais pas la même signature.

Qu'est ce que la signature d'une méthode ?
C'est le type de la valeur retournée et le type des paramètres qu'on trouve dans la déclaration[xv] :
`Type_retourné nom (Type nomPar1, Type nomPar2 ,...)`

[xv] En C , c'est ce qu'on appelle le prototype d'une fonction.

Intérêt : Permet l'abstraction, c'est-à-dire qu'une même action peut être implémentée de plusieurs manières tout en conservant le même nom.

3.2.9. Syntaxe de la déclaration d'un constructeur

[Modificateur] ***`NomClasse (paramètres) {`***
corps du constructeur (semblable à celui d'une méthode)
`}`

Syntaxe différente d'une méthode : Ce n'est pas une méthode ... c'est un constructeur !
Ainsi on ne peut appeler un constructeur qu'en utilisant `new`.
Remarque : Si vous définissez une valeur de retour , il s'agit d'uneméthode même si le nom est celui de la classe.

Tant qu'aucun constructeur n'est défini , il existe un constructeur par défaut qui ne produit aucune initialisation. Syntaxe : *`NomClasse ()`*

Modificateurs:
- Niveaux d'accès (encapsulation) : `private` , `protected` , `public`. Le niveau par défaut est appelé niveau paquetage (sans mot clé).

On utilise souvent la surcharge pour les constructeurs, ce qui permet différents types d'initialisations.

3.2.10. Encapsulation : Le retour

Les quatre niveaux d'accès définissent les ensembles de classes qui ont accès aux membres d'une classe.
- `private` : Accès autorisé seulement aux objets de la même classe.
  Utilisation : Pour des éléments "secrets" dont la modification risque de rendre l'objet inutilisable.
  Attention , pas de transmission par héritage. Usage rare.
- `protected` : Accès autorisé aux objets de la classe , des classes filles et des classes du même paquetage.
  Utilisation : Pour les éléments non publics a transmettre par héritage.
  Remarque : Les classes filles qui appartiennent à **un package différent** héritent des membres protégés mais **ne peuvent accéder** aux membres de leur classe mère.
- `public` : Accès autorisé à tous les objets.
  Utilisation : Niveau des membres permettant la communication entre les objets.
  A éviter pour les attributs. Conseillé pour les méthodes.
- package : Accès autorisé seulement aux objets de la classe et de même package.
  Remarque : Il ne s'agit pas d'un mot clé , c'est le niveau d'accès par défaut.
  Utilisation : **Usage rare.**

On remarque que les objets d'une même classe ont tous accès à tous leurs membres.

Réflexion:
- Quelle est la signification de ces quatre niveaux pour les constructeurs[xvi] ?

[xvi] Private: aucune classe ne peut instancier cette classe , il faut le faire à travers les méthodes publiques de la classe // Protected permet l'instanciation par les sous-classes etc...

3.2.11. Redéfinition des méthodes et polymorphisme

Les classes filles peuvent redéfinir toute méthode (ou attribut) héritée (sauf celle `final`).
On dit alors qu'on a écrit une méthode redéfinie, la méthode de la classe mère est dite cachée.

Syntaxe: Une méthode est redéfinie si les trois conditions suivantes sont remplies[xvii]:
- Le type de la valeur retournée est identique à la méthode cachée.
- Le nom de la méthode est le même (!).
- Le nombre et les types des paramètres sont identiques à la méthode cachée.

Autrement dit, si la signature de la méthode est la même

Question: Lorsqu'une de ces conditions n'est pas vérifiée comment la méthode est-elle interprétée[xviii]?

Remarques importantes:
- Les objets des **classes filles** sont considérés de **même type** que les objets de la **classe mère** (l'inverse est faux).
- Les méthodes redéfinies **peuvent présenter des modificateurs différents** de la méthode cachée.
- Une méthode `final` ne peut être cachée.

L'accéder aux méthodes (ou attributs) cachées se fait par le mot clef `super` : référence à la classe mère.

- Syntaxe: `super.`*`nomDeLaMéthode`* `(`*`paramètres`*`);`

3.3. Abstraction de la communication par messages : Interfaces.

3.3.1. Classes abstraites

Intérêt: Permettre la représentation d'un concept unificateur.

Modificateur: `abstract`.
Ex:
```
abstract class Nom {
     Corps de la clasee
     }
```

Une classe abstraite contient normalement au moins une **méthode abstraite** : C'est une méthode dont on ne donne pas l'implémentation.

Syntaxe d'une méthode abstraite:

`abstract` [*Modificateur2 ...*] ***`type_val_retournée nom (paramètres)`*** `[throws` *`exceptions`* `];`

Une classe abstraite ne peut être instanciée (on ne peut créer des objets) !

[xvii] C'est-à-dire qu'elles ont la même signature !
[xviii] c'est une surchare.

Seules les classes filles peuvent l'être à condition qu'elles ne soient pas elles mêmes abstraites (il faut qu'elles implémentent les méthodes abstraites héritées).
Mais une variable de type abstrait peut être passée en paramètre ou être l'attribut d'une classe! Le compilateur sait qu'à l'exécution c'est en fait un objet d'une sous classe qui sera le paramètre ou l'attribut (mais que cet objet respectera les spécifications de la classe abstraite).

EXEMPLE: On pourrait créer une classe abstraite appelée `Courbe` qui contiendrait une méthode abstraite `calcule` , elle formerait un concept unificateur pour la classe `Hexagone` , et pour toutes les autres classes permettant de tracer des courbes en 2D.

```
public class Courbe extends Polygon {
     abstract void calcule();
}
```

La classe `Hexagone` pourrait alors être une classe fille de `Courbe`.

3.3.2. Interfaces

Intérêt: Permettre la représentation d'un protocole de communication entre plusieurs classes.

Mot réservé pour l'interface (modèle de protocole): `interface`.
Syntaxe: `[public] interface` *Nom* `[extends` *Interface1,Interface2,...]{*
Corps de l'interface
}
Description: Une interface est un ensemble de définitions de méthodes et de constantes.
Ex: `[`*Modificateur2 ...*`]` *type_valeur_retournée nom* `(`*paramètres*`)`
`[throws` *exceptions* `];`

Une interface *ressemble* donc à une classe dont toutes les méthodes seraient abstraites. Mais:
- les méthodes sont par défaut `abstract`
- les constantes sont par défaut `public static final`.

Mot réservé pour la classe qui peut communiquer suivant le protocole: `implements`.
Syntaxe: `class` *Nom* `implements` *NomInterface1 , NomInterface2 ,...* `{...}`
Remarques importantes:
- Les interfaces peuvent hériter d'une interface (les hiérarchies d'interface sont possibles).
- Une classe peut implémenter plusieurs interfaces (ce qui n'est pas le cas pour l'héritage).

Exemple:

Si un certain nombre de classes sont amenées a utiliser des classes du type `Courbe`, mais aussi d'autres ayant en commun la méthode calcule(), on pourrait créer une interface qui s'appellerait `Tracé2D` par exemple.

```
public interface Tracé2D {
     void calcule();
     void trace();
}
```

4. Les composants Java

4.1. Qu'est ce qu'un composant ?

4.1.1. Définition:

C'est un terme général de l'informatique moderne qui correspond à un programme présentant les caractéristiques suivantes:

- Il remplit une fonction bien définie et est documenté pour permettre son utilisation sans avoir à connaître l'implémentation. D'ailleurs le source est souvent gardé secret par le concepteur du composant.
- Il sait communiquer avec les autres composants de son espèce.
- Il constitue une des parties de taille variable d'une application complète.
- Il possède les caractéristiques d'une classe d'objets.

4.1.2. Intérêt:

- Un programmeur peut développer toute une application en utilisant uniquement des associations de composants. Il n'est plus utile de connaître le langage dans toute sa finesse.
- Les applications deviennent complètement modulaires , on peut donc les reconfigurer simplement.

Un **bon composant** doit donc:

- Etre facilement configurable (mise en forme du composant par le programmeur)
- Fonctionner de manière concurrente (sans gêner les autres composants)
- Etre transposable d'une machine à une autre.

4.1.3. JavaBeans (grains de café):

Toute classe Java est potentiellement un JavaBean. Il faut donc se poser la question suivante pour savoir si la conversion en JavaBean est utile : Est ce que la réutilisation du code sera facilitée par cette transformation ?

a. Structure d'un Bean

Il présente des propriétés , des méthodes et de évènements:

- Les propriétés sont les attributs de la classe possédant des accesseurs (§3.2.6). Ceux sont les propriétés qui permettront la configuration du Bean au moment de l'assemblage.
- Les méthodes sont les méthodes publiques d'une classe. Elles permettront les actions dynamiques sur le Bean.
- Les évènements permettent d'avertir les autres éléments de l'application que quelque chose s'est produit. Le modèle d'événement est celui des classes (§4.3).

b. Possibilités d'un JavaBean

- Introspectable : Il est possible de découvrir à l'exécution les caractéristiques internes d'un Bean.
- Persistance : Un bean paramétré peut sauvegardé son état et le restaurer à l'exécution.
- Auto-installation : Un Bean livré dans un fichier archive saura être utilisé automatiquement dans l'application.

Avant de rentrer plus en détail dans la conception des JavaBeans (enTD par exemple), il nous faut présenter deux notions utilisées surtout dans le cadre des composants: Les classes internes et les évènements.

4.2. Les classes internes

*Afin de pouvoir développer un programme complexe **à l'intérieur** d'une classe, et donc dans un JavaBean , il est possible depuis le JDK (Java Development Kit) 1.1 de créer des classes internes.*

4.2.1. Définition

Une classe interne est tout simplement définie à l'intérieur d'une autre classe (dite **englobante**).
Ex:

```
public class Grain {
class PetitGrain {
     ...
     }
}
```

- Elle peut être **définie** soit comme attribut, soit à l'intérieur d'une méthode.

4.2.2. Autres propriétes

- Sa **portée** est semblable à une variable définie dans les mêmes conditions.
- Une classe interne a une **accessibilité totale** sur les membres de la classe englobante.
- A l'interieur d'une classe interne , le mot clé `this` peut faire référence, soit à l'instance courante de la classe interne, soit à celle de sa classe englobante (syntaxe *`nom_classe.this`*).
- On peut définir des **classes anonymes** , c'est à dire sans nom , elles sont alors utilisées dans des expression , par exemple en tant que paramètre. Une telle classe doit implémenter une interface ou dériver d'une classe.

Voici un exemple:
Supposons qu'on crée l'interface suivante:

```
Interface Affiche {
public void affiche (String chaîne);
}
```

On peut alors à l'intérieur d'une classe **instancier l'interface sans donner de nom à la classe interne**, ce qui permet *par exemple* à une méthode de renvoyer une instance de la classe sans alourdir le code avec la création d'une classe interne complète:

```
return new Affiche () {
     public void affiche (String chaîne) {
     System.out.println (chaîne);
     };
```

Explication du code ci-dessus : la classe entre {...} implémente l'interface Affiche. Elle est instanciée immédiatement (`new`) c'est le paramètre renvoyé par une méthode à travers d'un `return` , d'où la nécessité du ; qui désigne la fin de l'instruction `return`.

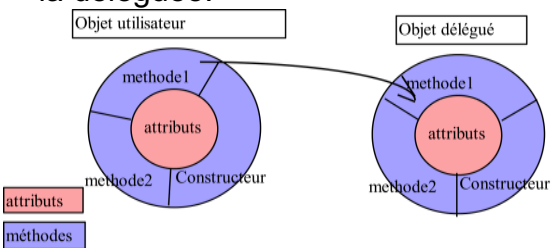
Conseil : ce code est difficile à lire, n'utilisez les classes anonymes que dans de *petites classes*.

- Une classe interne peut être un attribut `static` de la classe principale: Une telle classe se comporte comme une classe principale mais qu'on ne peut utiliser qu'à travers sa classe englobante. Permet de consolider le lien entre deux classes. (peu utilisé)

4.2.3. Principale utilisation : l'adaptation

L'adaptation est une notion liée à la délégation.

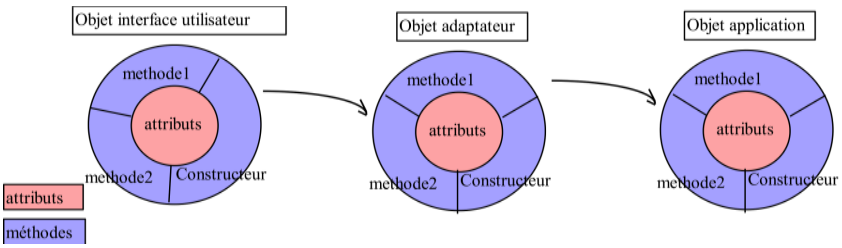
- La **délégation** est une notion souvent utilisée en programmation objet : Il s'agit de rajouter des fonctionnalité à une classe en utilisant des méthodes appartenant à un autre classe. C'est à dire qu'une classe **délègue** à une autre classe une partie de ses fonctionnalité. Cette autre classe est la déléguée.



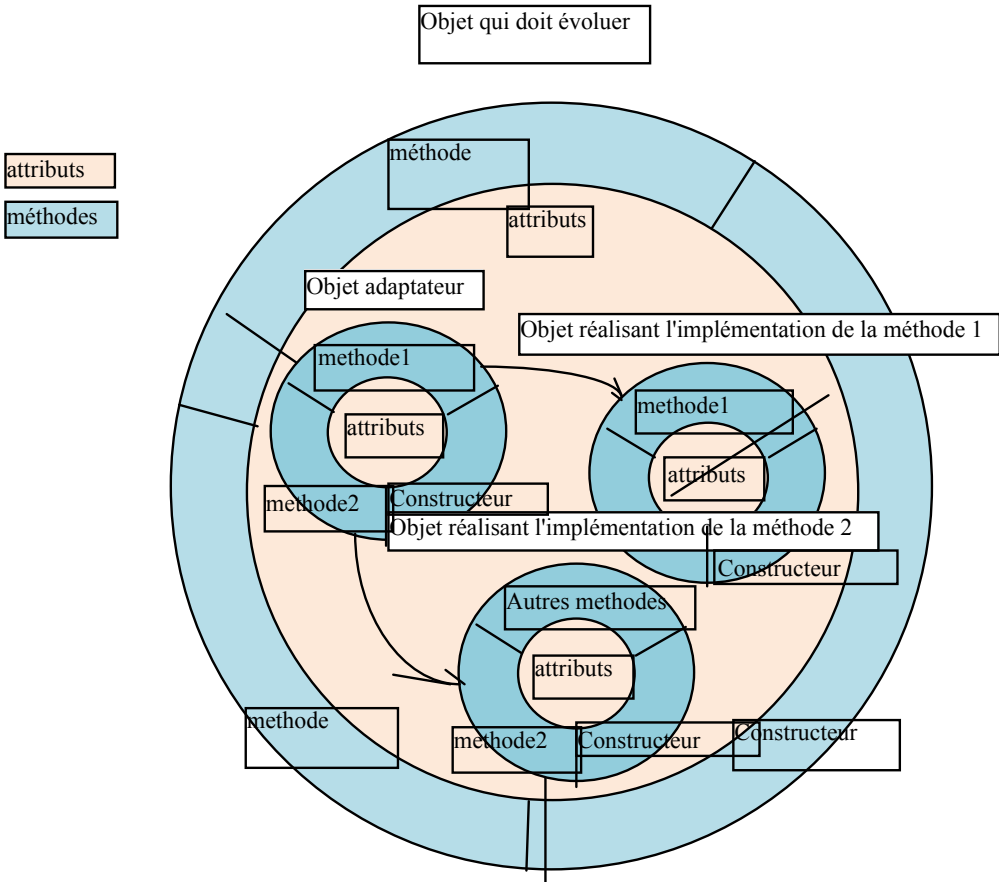
Pourquoi cette notion ? Pour répondre à la situation suivante : on est en présence d'une classe figée qui a besoin de nouvelles fonctionnalité. Une **classe figée** est une classe qu'on ne désire pas ou qu'on ne peut pas la modifier :

- Qu'est-ce que l'**adaptation** ? C'est une version plus complexe de délégation qui est nécessaire dans le cas ou les deux classes (utilisatrice et déléguée) sont figées. On utilise pour régler ce problème un **adaptateur**. Il s'agit d'une classe qui sera chargée de jouer le rôle de tampon entre la classe figée et la classe déléguée figée.

Par exemple chacune des deux classes à adapter correspondent à deux concepts qu'on désire isoler l'un de l'autre. C'est le cas d'une classe (nommée application) qui effectue un certain nombre de tâches qu'on désire utiliser dans plusieurs programmes. Pour mettre en œuvre ces fonctionnalités dans un certain programme il est nécessaire de mettre au point une interface utilisateur qui sera une classe indépendante. Dans ce cadre la classe interface utilisateur délègue à la classe application son fonctionnement, c'est un cas de **délégation**. Mais supposons maintenant qui l'interface utilisateur soit récupérée d'une ancienne version du programme (on ne veut pas la modifier), il faudra intercaler une classe **adaptatrice** entre les deux.



Autre exemple, une des classes est déjà utilisée par d'autres classes , vous ne pouvez donc pas changer sa signature , mais vous voulez maintenant qu'elle implémente une nouvelle interface. De plus les méthodes qui implémenteront la nouvelle interface existent déjà dans une (ou des) classe séparée.



Maintenant que cette notion a été introduite, quel rôle vont jouer les classes internes dans ces cas?

Le premier exemple correspond à la gestion des événement (O combien indispensable) que nous traiterons en détail dans le chapitre suivant.

Le second pourrait être traité de la façon suivante :

```
// Les classes existantes qui implémentent les méthodes manquantes
class ImpMethode1 {
    ...
    public void methode1 {
    ...
    }
}

class ImpMethode2 {
    ...
    public void methode2 {
    ...
    }
}
// L'interface
interface MonInterface {
    public void methode1();
    public void methode2();
}

// La classe figée
```

```
class Figée {
    private ImpMethode1 impMethode1=new ImpMethode1();
    private ImpMethode2 impMethode2=new ImpMethode2();

     // Utilisation d'une classe anonyme pour faire l'adaptateur
    private MonInterface adaptateur=new MonInterface() {

    public void methode1() {

    ImpMethode1.methode1();
    }

    public void methode2() {

    ImpMethode2.methode2();
    }
    };

    suite de la classe figée...
}
```

C'est la classe anonyme permettant de créer un objet qui implémente l'interface qui joue le rôle d'adaptateur. Un objet de cette classe est un attribut de la classe figée. On pourra donc l'utiliser quand ce sera nécessaire de façon à ce que la classe figée présente les caractéristiques imposées par l'interface à travers cet objet et sans implémenter directement l'interface.

Bien sûr on aurait pu traiter ce problème avec un adaptateur externe, mais cela aurait alourdi le code inutilement.

4.3. Les événements

4.3.1. Principe de la gestion événementielle

1. Comment est matérialisé l'événement ?

   Un objet d'une classe réagissant à des événements (par exemple une classe réagissant au clic d'une souris si il s'agit d'une fenêtre) **crée un objet d'une la classe de type événement, c'est-à-dire une instance de classe issue de la hiérarchie d'événements fournie par l'AWT** (Abstract Window Toolkit).

2. Qui récupère cet événement (ou objet d'une certaine classe d'évènements) ?

   Les classes générant des événements possèdent des **méthodes permettant d'indiquer à quel objet transmettre l'événement**. Ces méthodes sont normalisées par l'AWT:

   ```
   public type_evenementListener
   settype_evenementListener(type_evenementListener nom);
   ```

   ou

   ```
   public type_evenementListener
   addtype_evenementListener(type_evenementListener nom);
   ```

type_événement fait partie du nom de la méthode et précise **quelle action de l'utilisateur** met en œuvre le processus de gestion événementielle.

nom est le nom de **l'objet qui reçoit l'événement**, on lui donne le nom de ***listener***. Cet **objet doit implémenter l'interface *type_evenementListener*** qui est définie dans la hiérarchie des `EventListener`. On appelle un tel objet un **listener**[xix].

Ces méthodes sont habituellement placées dans le constructeur.

3. Comment est traité l'événement ?

L'objet **listener** (c'est-à-dire qui implémente l'interface ***type_evenement*`Listener`) destiné à traiter l'événement **possède obligatoirement la méthode suivante**:

```
public void type_evenement_effectue (type_evenementEvent e) {...}
```

(imposée par l'interface correspondant à l'événement traité[xx]).

Cette méthode est automatiquement appelée au moment ou l'événement est produit, et l'objet événement correspondant lui est transmis en paramètre. Ainsi dans cette méthode ont peut exploiter tous les renseignements contenus dans l'objet événement. **C'est donc dans cette méthode que le traitement de l'événement se déroulera.**

4.3.2. Classes et interfaces liées à la gestion événementielle

Les classes événements sont les classes filles de `java.util.EventObject` par l'intermédiaire de `java.awt.AWTEvent`.

On peut donc définir ses propres événements en héritant de `EventObject`.

Les interfaces listeners héritent de `java.util.EventListener`

4.3.3. Plusieurs façons de coder

a. La façon la plus courte (délégué interne) : méthode par défaut de WindowBuider pour Eclipse

```
public class UI {
    // l'interface utilisateur
    public UI(){
    ...
    Button bouton1=new Button();
    bouton1.addActionListener(

    new ActionListener() {
              // classe anonyme déléguée
```

[xix] Pour revenir aux notions de délégation , on s'aperçoit que cet objet va jouer le rôle de délégué. En effet vous allez voir au point 3 que c'est lui qui fourni la méthode pour traiter l'événement.

[xx] Parfois certaines interfaces imposent l'implémentation de nombreuses méthodes , pour simplifier la tâche il existe des délégués par défaut dont peut hériter votre délégué. Il faudra alors simplement redéfinir la méthode correspondant à l'action qui importe pour vous. Ces délégués se trouvent dans le paquetage `java.awt.event.` Les méthodes des adaptateurs par défaut sont vides , on ne peut les utiliser directement.

```
    public void actionPerformed (ActionEvent e) {

    ...// traitement de l'événement

       }
    }
    );
    ...
    }
...
}
```

Cette façon de traiter les événement est pratique pour les petites applications : Le code est concentrer et l'utilisation de la casse anonyme rapproche au maximum le traitement de l'événement de la création de l'objet générant les événements.

On reconnaît une classe anonyme qui peut jouer le rôle de **délégué** si elle effectue elle même le traitement de l'événement (c'est un cas extrême de délégation, ou la délégation ne présente aucun intérêt). Elle joue le rôle d'**adaptateur** si elle fait appel à une méthode d'une autre classe pour traiter l'événement.

b. Une façon plus perfectionnée pour les grandes applications

Pour les grandes applications (ici 2 boutons!), ou celles où on veut distinguer clairement la partie interface utilisateur de la partie application (c'est l'exemple 1 du début) , il vaut mieux utiliser un adaptateur (interne ou externe) à part entière , c'est à dire concentrant tous les traitements .

Voici un exemple de réalisation.

```
public class UI {
    // l'interface utilisateur
    public UI(){
    ...
    Button bouton1=new Button();
    bouton1.addActionListener(new Adaptateur(1)); // enregistrement
    Button bouton2=new Button();
    bouton2.addActionListener(new Adaptateur(2)); // enregistrement

    ...
    }
...
}

class Adaptateur implements ActionListener() {
    // classe adaptatrice


    int aiguillage;// va permettre l'aiguillage des événements

    Délégué délégué=new Délégué(); // le délégué !
```

```
    public Adaptateur (int aiguillage) {

    this.aiguillage=aiguillage; // initialisation d'aiguillage
          }


    public void actionPerformed (ActionEvent e) {

    switch (aiguillage) {

    case 1:
    délégué.méthode1();


    break;

    case 2:
    délégué.méthode2();


    break;
    }
    }
}

class Délégué{
    // c'est la classe implémentant les actions

    ....
    .... méthode1() {
    ....
    }

    ....méthode2() {
    ....
    }
    ....
}
```

Remarque: Si vous avez besoin de plusieurs délégués, au lieu de les créer tous avec votre objet adaptateur, il vaudrait mieux les créer suivant vos besoins dans le constructeur de l'adaptateur.

On modifiera alors le code de l'adaptateur de la façon suivante :

```
class Adaptateur implements ActionListener() {
    // classe adaptatrice


    int aiguillage;// va permettre l'aiguillage des événements

    Object délégué;
    // le délégué !
```

```
    public Adaptateur (int aiguillage) {

    this.aiguillage=aiguillage; // initialisation d'aiguillage

    switch (aiguillage) {

    case 1:
    délégué=new Délégué1();


    break;

    case 2:
    délégué=new Délégué2();


    break;

    }


    public void actionPerformed (ActionEvent e) {

    switch (aiguillage) {

    case 1:
    délégué.méthode1();


    break;

    case 2:
    délégué.méthode2();


    break;
    }
}
}
```

Bien sûr, les délégués devront implémenter les méthodes 1 et 2.

5. Gestion des exceptions

5.1. Qu'est ce qu'une exception ?

Lorsque la machine virtuelle se trouve face à une erreur elle peut réagir de deux façons :
- Il s'agit d'une erreur fatale, le programme est arrêté.
- Il s'agit d'une erreur qui peut être gérée, une **exception** , elle fabrique un objet renseignant sur l'**exception**. Cet objet hérite alors de la classe `java.lang.Throwable.exception`

Cette exception doit être gérée soit par votre programme , soit par la machine virtuelle qui met un terme au programme... Il vaut mieux que ce soit votre programme qui la gère.

5.2. Le mécanisme de propagation des exceptions

Une exception peut être lancée (thrown en anglais) par la machine virtuelle , ou par votre programme grâce au mot réservé `throw` , voir plus loin.

Une fois une exception lancée le déroulement normal du programme est stoppé. Le programme saute au bloc destiné à l'attraper (bloc `catch` ... toujours en anglais). Si ce bloc n'existe pas dans la méthode ou est lancée l'exception , le programme saute au bloc `catch` de la méthode appelante etc... Si l'exception n'est toujours pas attrapée après `main()` la machine virtuelle la gère en arrêtant le programme. C'est le mécanisme de **propagation** des exceptions.

5.3. Attraper une exception : Le bloc `catch` , Le bloc `finally`

Syntaxe:

Dans la méthode qui veut attraper l'exception on place la structure suivante :

```
... methode (...) {
     ...// début du code de la méthode
     try {//bloc de code susceptible de lancer une exception
     ...
     }
     ...// fin du code de la méthode

     catch (typeException nomException)                                    {
     // début du traitement
     ...
     }// fin du traitement
     ...// il peut y avoir d'autres blocs catch
}   // fin de la méthode
```

Remarques:
- Les blocs traitent uniquement les exceptions qui correspondent au type ***typeException*** ou à une de ses classes mères.
- Les blocs `catch` sont parcourus dans l'ordre.
- Lorsque l'exception est traitée par un bloc `catch` , la méthode se termine sur ce bloc , sauf si un bloc `finally` se trouve dans la méthode.

A la suite des blocs `catch` ou à leur place on peut placer un bloc `finally` , ce bloc sera exécuté dans tous les cas d'exception.

5.4. Lancer des exceptions : `throw` et `throws`

- Pour **lancer** une exception:

Dans votre méthode il suffit de placer à l'endroit approprié

```
... throw new typeException ();
```

Vous êtes alors **obligé d'attraper l'exception** ! Sauf si vous voulez la propager ...

- Pour **propager** une exception:

C'est le cas lorsqu'on ne veut pas traiter l'exception dans la méthode qui la lance ou lorsqu'on utilise une méthode qui propage elle même une exception.

Syntaxe:

```
... nomMethode ... throws typeException (){
...
}
```

Remarque : Lorsque vous utilisez une méthode qui propage une exception vous êtes **obligé d'attraper l'exception ou de la propager , sauf** pour les exceptions de type `RuntimeException` (ou qui en hérite) afin d'éviter d'alourdir le code (ces exceptions sont les plus courantes , elles sont lancées pour des expressions qui ne sont pas correctement évaluées ou des indices de tableaux qui sortent de leurs limites).

- Pour créer ses propres exceptions:

Il faut hériter de `java.lang.Throwable.exception`

Exemple:

```
class MonException extends Exception {
     public MonException () {
          super();
     }
     public MonException (String s) {
     super(s);
     }
}
```