D'Alice à Java

version 0.3 du 10/09/10

David Roche

Notes de l'auteur :

Après une première approche de la Programmation Orientée Objet (POO) avec Alice (voir "Apprendre la POO avec Alice" disponible sur www.animanum.com/alice), certains élèves du lycée Guillaume Fichet (Haute-Savoie) nous ont fait part de leur désir de poursuivre leur découverte de la POO en classe de 1èreS en s'initiant à Java.

Le but de ce document est de donner aux élèves les notions de base qui leur permettront d'aborder l'élaboration de petites applications sous le système d'exploitation de l'open handset allience : Android (http://www.openhandsetalliance.com/)
Beaucoup de notions ne sont pas abordées dans ce document, des notions pourtant importantes, mais je pense que trop de théorie risquerait de "décourager" un trop grand nombre d'élèves.
Ce document n'a donc pas la prétention d'être une "Bible du Java" (il existe déjà beaucoup d'ouvrages de qualités sur Java). J'ai essayé d'établir une passerelle (pas trop difficile à franchir pour des élèves de lycée) entre les notions vues l'année dernière avec Alice et la « vraie » programmation en Java.

Les tableaux, les exceptions et les threads seront introduits quand cela sera nécessaire dans le 3ème document consacré à la programmation sous Android : « De Java à Android » (titre provisoire)

David Roche, août 2010

Modifications version 0.1 à 0.2

- Ajout dans le chapitre II : "surcharge d'une méthode" et "méthodes et champs static"
- Écriture chapitres III,IV,V,VI et VII (fin)
- Mise en page

Modifications version 0.2 à 0.3

modifications suite aux commentaires de Thierry Vieville, chercheur à l'INRIA (un grand merci à lui) :

- Suppression de la partie do/while
- autres modifications (vocabulaire employé, convention de nommage,.....)

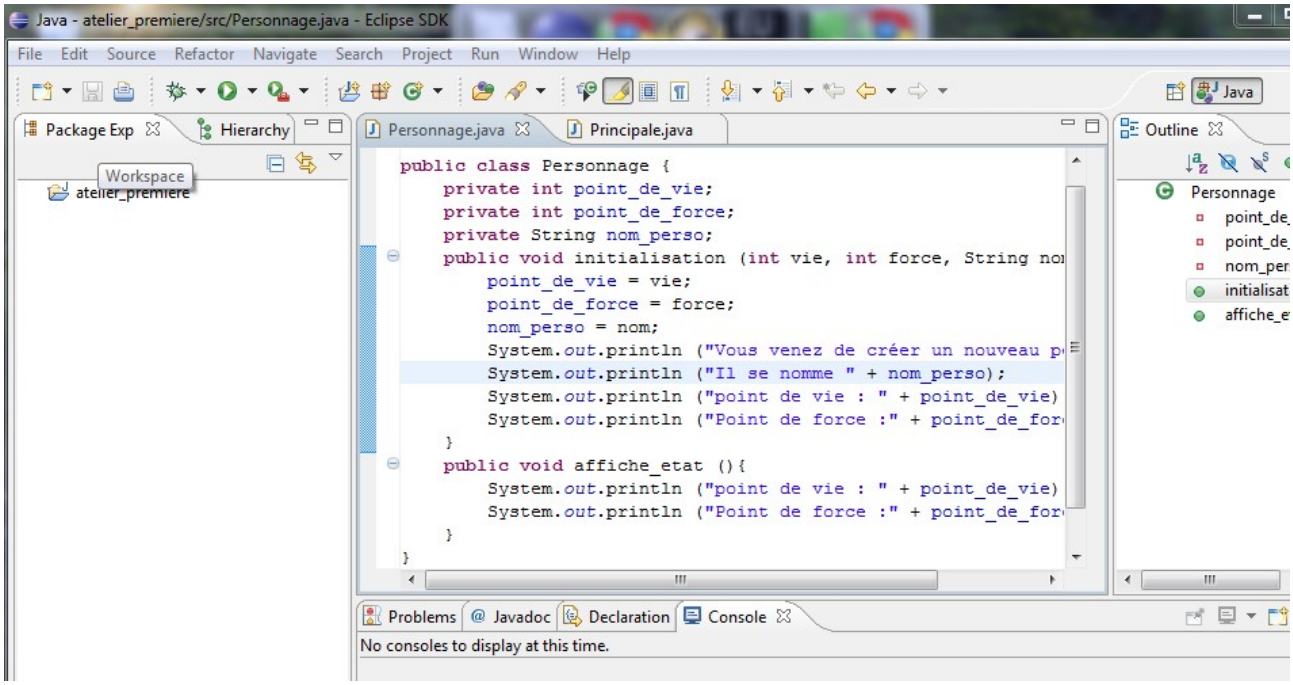
Sommaire

Chapitre 1
Introduction

L'utilisation du programme Alice nous a permis de nous initier aux bases de la programmation orientée objet (voir « Apprendre la POO avec Alice »). Nous allons maintenant passer à l'étape suivante : l'apprentissage d'un « vrai » langage: le Java.
Java a été développé en 1995 par des ingénieurs de la société SUN Microsystems, pour piloter de petits appareils électriques. Aujourd'hui (nous en sommes à la version 6), Java est utilisé dans de nombreux domaines de l'informatique : création web, développement d'applications pour les smartphones (iphone, « androphone »,...) et pour bien d'autres choses. L'universalité de Java tient, entre autres, à l'utilisation d'une machine virtuelle qui permet de s'affranchir des contraintes liées au système d'exploitation (un programme java écrit pour Windows pourra être, à peu de choses près, utilisé tel quel sous linux. C'est moins le cas pour d'autres langages, comme par exemple le C++).

Pour programmer en Java, nous allons utiliser un IDE (qui signifie ici *Integrated Development Environment* (Environnement de développement intégré)) : Eclipse (http://www.eclipse.org/)
NB :

- l'utilisation d'un IDE n'est pas obligatoire (juste recommandé !), il est tout à fait possible d'utiliser un simple éditeur de texte pour écrire du code.
- Il existe d'autres IDE (par exemple netbeans), cela sera à vous de faire votre choix



Pour commencer, il faut créer un nouveau projet : file -> New -> Java Project

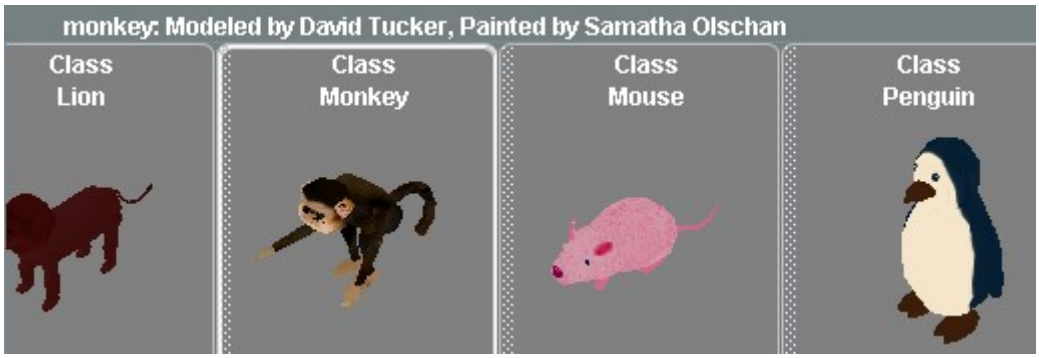
Une fois le nouveau projet créé, il faudra créer une classe : file -> New -> Class

Chapitre 2
Les classes, les champs et les méthodes

Beaucoup de notions introduites avec Alice vont se retrouver dans Java : les classes, les instances, les méthodes, les variables. Nous allons bien sûr, retrouver aussi les "if/else", le "while" et bien d'autres choses !

Création d'une nouvelle classe

Dans Alice, toute une série de classes vous est proposée :



À partir de ces classes, vous pouvez créer des instances de classes (voir le chapitre IV (1ère partie) d' «Aprendre la POO avec Alice » pour plus de précisions sur la notion d'instance de classe) . Chaque instance possède alors des méthodes, des fonctions et des variables d'instance (properties dans Alice). Avec Alice, nous avons utilisé des classes préexistantes, il existe aussi beaucoup de classes « toutes prêtes » en Java, et nous aurons l'occasion d'en utiliser à l'avenir, mais pour commencer notre découverte de Java, nous allons apprendre à créer une classe de a à z.

Pour créer une nouvelle classe, il faut commencer par la déclarer :

Ex 2.1

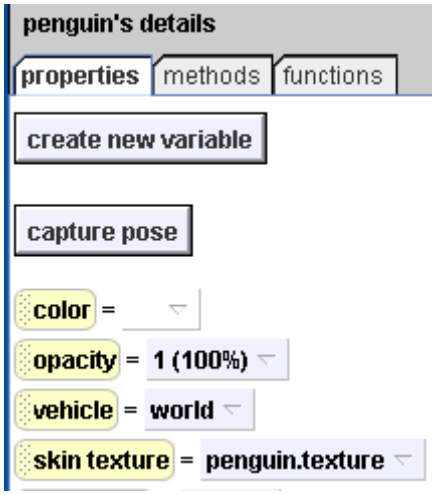
```
public class Personnage {

}
```

Nous avons créé ici une classe nommée Personnage (remarquez tout de suite la majuscule, c'est un usage qu'il faut respecter : Personnage). J'attire aussi votre attention sur l'utilisation du mot clé *class*, comme dans Alice (voir un peu plus haut : *class* Monkey). Le mot *public*, signifie simplement que cette classe sera utilisable par « tout le monde » (on aurait pu mettre à la place de *public*, *abstract* ou *final*, mais nous n'aborderons pas ce sujet dans ce cours). Dernière chose, les 2 accolades ({ ouvrant et } fermant) qui annoncent le début et la fin de notre classe (pour l'instant il n'y a rien entre les deux, mais cela ne va pas durer !)

Variables et champs

Dans Alice, nous avons déjà vu la notion de variable (onglet properties) :



Notre classe *penguin* possède des *properties*, par exemple *color* permet de définir la couleur du pingouin (ci-dessus, *color* = blanc). Avec Java, nous allons rencontrer exactement la même notion, les champs d'une classe.

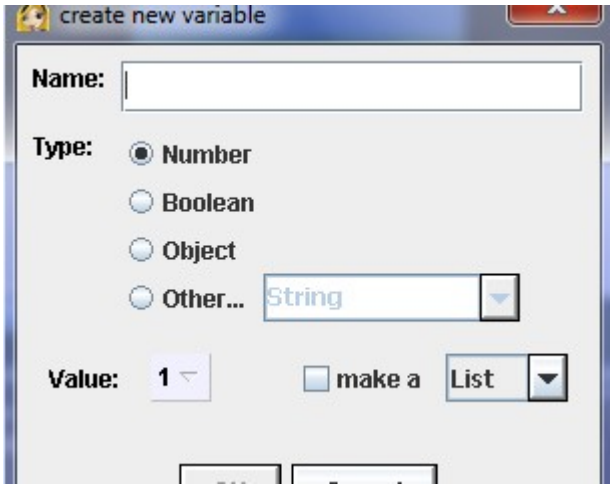
Ex 2.2

```
public class Personnage {
	int point_de_vie;
	int point_de_force;
}
```

Nous avons créé une classe Personnage, une instance de cette classe Personnage possédera des points de vie et des points de force (en cas de blessure, notre personnage devra perdre un point de vie ; un combat (et donc la fatigue qui va avec !) fera perdre à notre personnage des points de force)

Ces champs d'instance sont des variables, faisons donc une petite parenthèse sur l'utilisation des variables en Java :

Dans Alice, les variables sont dites "typées"



Au moment de la création d'une variable avec Alice, vous devez choisir entre : *Number* (nombre), *Boolean* (booléen), *Object* ou *other* (dans other on retrouve notamment *String* (chaîne de caractère)).
En java, si vous désirez créer une nouvelle variable, il faudra aussi indiquer quel est le type de votre variable (on parle de déclaration) :

int	Entier compris entre -2147483648 et 2147483647
boolean	true ou false
double	Décimaux entre $4{,}9.10^{-34}$ et $1{,}79769.....10^{308}$ (4.9E-324 et 1.79769.....E308)
char	1 caractère
String	Chaîne de caractères (plusieurs caractères)

NB 1: Pour être un peu rigoureux tout de même, il faut noter que *String* n'est pas un type de variable, mais une classe (mais nous l'utiliserons comme un type), d'où la majuscule !
NB 2 : Il existe d'autres types (float, long,...), mais nous utiliserons uniquement ceux cités ci-dessus
NB 3: Dans beaucoup de cas (comme ici), nous utiliserons "int". Nous verrons dans la suite du cours des exemples d'utilisation des autres types.

Voici quelques règles à respecter pour le nom des variables :
- le nom d'une variable doit commencer par une lettre
- il peut être constitué de lettres, de chiffres ou du caractère _ (voir exemple)
- il y a une distinction entre minuscules et majuscules (comme dans Alice)
- les mots clés du langage (à voir plus loin) ne doivent pas être utilisés comme nom de variable.

Avant de pouvoir utiliser une variable, il faut donc la déclarer en précisant le type :
[type] nom_de_la_variable ;

exemple : `int point_de_vie;`

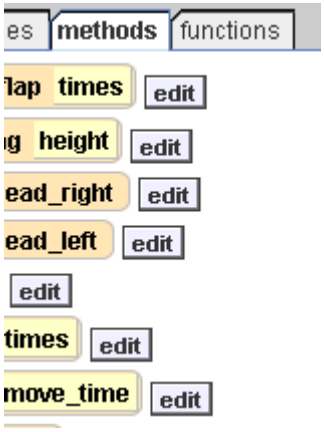
Autre chose que vous avez surement remarqué, le point virgule à la fin de la ligne.
Pour signifier au compilateur (programme qui transforme votre code en instruction compréhensible par l'ordinateur) que notre ligne est terminée, le simple retour chariot (appui sur la touche "Entrée") ne suffit pas, il faut utiliser un point virgule (essayez de le supprimer, Eclipse vous signifiera alors une erreur).
Ajoutons maintenant le mot clé *private* à nos 2 déclarations de variables :
Ex 2.3

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
}
```

Sans trop entrer dans les détails, le sujet est abordé un peu plus loin (voir « l'encapsulation »), l'utilisation de ce mot clé permet "d'interdire" toutes utilisations de ces champs depuis l'extérieur de la classe *Personnage*.

Les méthodes

Dans Alice, une classe possède aussi des méthodes et des fonctions (onglet *methods* et onglet *functions*) : les fonctions sont des méthodes qui renvoient des valeurs



En java, la distinction est moins évidente, puisque nous allons uniquement parler de méthodes. Mais les méthodes en java correspondent aux fonctions d'Alice (les méthodes de java renvoient une valeur à l'aide du mot clé *return*, comme les fonctions dans Alice !!) sauf quand le mot clé *void* est utilisé (voir ci-dessous) où là, la méthode ne renvoie rien (tout cela va s'éclaircir avec quelques exemples)

Voici la procédure pour créer une méthode :
[modificateurs] [type de retour] nom_de_la_méthode ([liste des paramètres])

[modificateurs] : private => la méthode ne peut être utilisée que dans la classe où elle est définie
public => la méthode peut être utilisée depuis n'importe quelle autre classe.
Il existe d'autres modificateurs (*protected*, *abstract*, *final*,.....) que nous ne verrons pas dans le cadre de ce cours.

[type de retour] : c'est le type (*int*, *String*,.....) de la valeur renvoyée par la méthode à l'aide du mot clé *return* (ce qui correspond aux fonctions dans Alice).
Quand la méthode ne renvoie aucune valeur, on utilise le mot clé *void* en remplacement du type de retour (cela correspond donc aux méthodes dans Alice).

Nous allons ajouter à notre classe Personnage une première méthode : *initialisation*.
Cette méthode sera utilisable en dehors de la classe (*public*) et elle ne renverra aucune valeur (*void*).
Dans un premier temps, cette méthode n'acceptera aucun paramètre (nous verrons un peu plus loin l'utilisation des paramètres en java).
Ex 2.4

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	public void initialisation (){
		point_de_vie = 100;
		point_de_force = 50;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
}
```

Avant d'étudier notre première méthode en détail, peut-être avez-vous déjà remarqué qu'Eclipse introduit parfois un décalage de quelques espaces d'une ligne à l'autre, cela s'appelle l'indentation. L'indentation est très souvent utilisée en programmation; elle a pour but de rendre le code plus lisible.
Comme pour une classe, notre méthode commence par une accolade ouvrante et se termine par une accolade fermante.
Les 2 premières lignes de notre méthode donnent une valeur au champ défini un peu plus haut dans le code, *point_de_vie* et *point_de_force*.

```
point_de_vie = 100;
point_de_force = 50;
```

Rien à signaler (il ne faut pas oublier les points virgules !)

Ensuite, une vraie nouveauté :

```
System.out.println ("Vous venez de créer un nouveau personnage");
```

Si vous êtes un peu malin (et je n'en doute pas !), vous aurez sans doute deviné que l'instruction `System.out.println` va nous permettre d'afficher quelque chose à l'écran, mais quoi ? Et bien tout simplement ce qui se trouve dans la parenthèse qui suit. Si vous voulez afficher une chaine de caractères, il faut la mettre entre guillemets : `("Vous venez de créer un nouveau personnage")`

En conclusion, cette ligne va nous permettre d'afficher à l'écran :

```
Vous venez de créer un nouveau personnage
```

Même chose pour les 2 lignes suivantes, enfin, presque :

```
System.out.println ("point de vie : " + point_de_vie);
```

seule nouveauté, le « + point_de_vie » va afficher la valeur de la variable *point_de_vie* (dans notre cas « 100 »).

Pour la dernière ligne de notre méthode, je pense qu'il n y a aucun problème.

Bon, tout cela est bien beau, mais comment va-t-on utiliser notre belle classe toute neuve ?

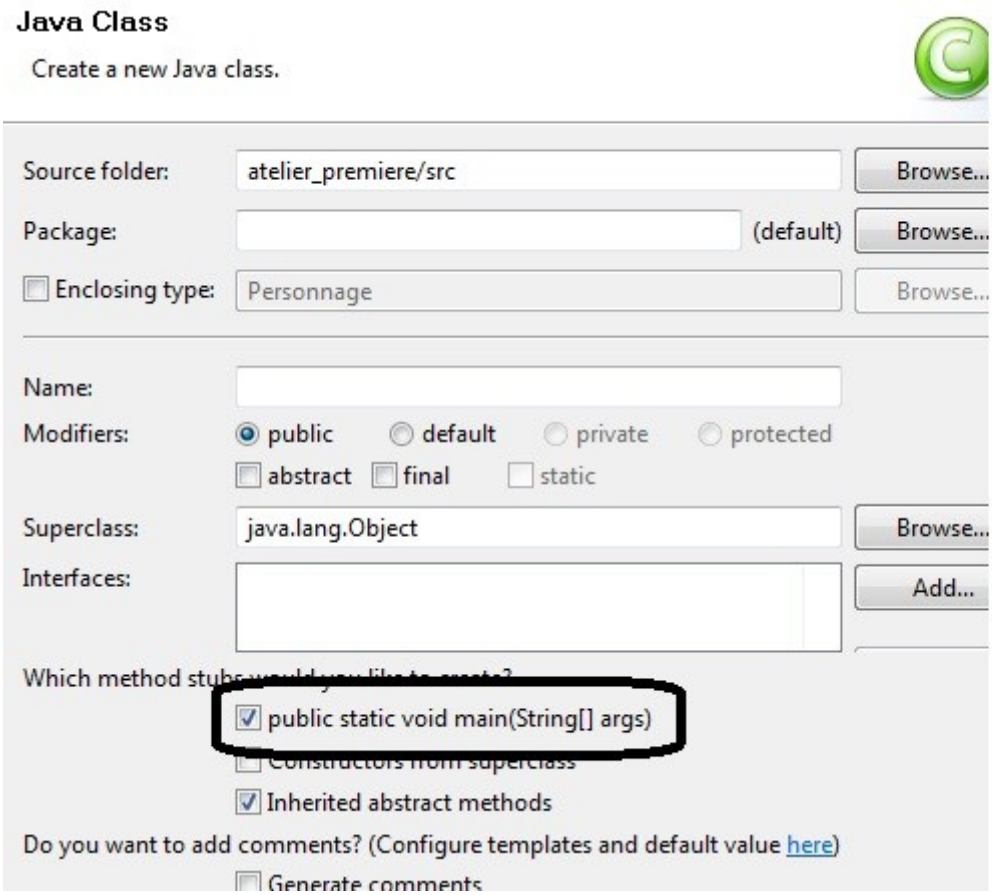
Nous devons créer une deuxième classe (nous allons la nommer *Principal*), dans Eclipse vous devez faire : file -> New -> Class

Mais pour que notre programme fonctionne, il faut que notre nouvelle classe possède une méthode un peu particulière : la méthode *main*

Qu'est-ce que cette méthode *main* ?

Presque tous les programmes en java en possèdent une, c'est la première méthode qui sera exécutée au lancement de votre programme; un programme sans méthode *main*..... ne fera rien !
Eclipse va nous permettre de créer notre méthode *main* automatiquement au moment de la création de notre classe *Principal.*

Au moment de la création de la classe *Principal*, vous devez cocher « public static void main..... » comme indiqué ci-dessous



Pour l'instant, les autres options nous importent peu, nous les laisserons donc telles quelles.
Dans Eclipse, on obtient alors notre nouvelle classe et la méthode *main* déjà paramétrée comme il se doit.

Ex 2.4 (suite)

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
```

La méthode *main* doit être publique (*public*), et ne doit pas renvoyer de valeur (*void*). Le mot clé *static* est obligatoire, tout comme le *(String[] args)*.

Création d'une instance

Dans Alice, pour créer une instance de classe, il suffit de faire un "cliquer, glisser et déposer".
Avec java, c'est presque aussi simple :
Créons une instance de notre classe *Personnage;* nous appellerons notre instance *bilbo*

La première chose est de déclarer *bilbo* comme étant de type *Personnage* (un peu comme avec les variables) :
Personnage bilbo
Pour l'instant, notre instance *bilbo* n'existe pas encore, l'ordinateur a seulement réservé une petite place en mémoire pour accueillir une instance de *Personnage*.

Pour terminer la création de notre instance, il faut écrire :

bilbo = new Personnage ()

Souvent, on regroupe les 2 lignes précédentes :

Personnage bilbo = new Personnage ()

Voici ce que cela donne dans Eclipse.

```
Ex 2.5

public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		}
}
```

Appliquer une méthode à une instance

Maintenant que notre instance *bilbo* est créée, nous allons pouvoir l'utiliser. Nous allons lui appliquer notre méthode *initialisation*. Le code est très simple :
nom_de l'instance.nom_de la méthode
Ce qui donne pour notre instance *bilbo* et notre méthode *initialisation* :

ex 2.6

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		bilbo.initialisation();
		}
}
```

Résultat :

```
Vous venez de créer un nouveau personnage
point de vie : 100
Point de force :50
```

Utilisation des paramètres

Dans Alice nous avons, à de nombreuses reprises, utilisé les paramètres dans les méthodes

Dans l'exemple ci-dessus, nous avons une instance de la classe Hare. J'ai créé une méthode *saut_lievre* qui fait sauter notre lièvre ! Le lièvre adorant sauter, il saute toujours plusieurs fois dz suite. Mais combien de fois ? Et bien, nous avons créé un paramètre de type *number* : *nbre_de_saut*. Je vous laisse revoir cet exemple par vous-même, cela ne devrait pas vous poser de problème (ici, notre lièvre sautera 2 fois)

Et en Java alors ?

C'est exactement le même principe, nous allons créer 2 paramètres : *vie* et *force*. Comme dans Alice, il est obligatoire de fournir le type des paramètres, ici des entiers, on aura donc :
int vie et *int force*.
Dans Alice, pour créer des paramètres dans une méthode, il suffit de cliquer sur le bouton « create new parameter ». En Java ce n'est pas beaucoup plus compliqué; pour notre classe *initialisation* on aura :

public void initialisation (int vie, int force) {

Voilà, nos paramètres de la méthode *initialisation* sont en place !

Voici notre classe modifiée :

ex 2.7

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	public void initialisation (int vie, int force){
		point_de_vie = vie;
		point_de_force = force;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
}
```

Ensuite nous utiliserons nos 2 paramètres pour attribuer des valeurs à nos 2 champs, *point_de_vie* et *point_de_force* :

```
point_de_vie = vie;
point_de_force = force;
```

Mais comment faire passer nos paramètres ?

Tout simplement au moment de l'appel de la méthode :

nom de l'instance.nom de la méthode (valeur paramètre 1, valeur paramètre 2,.....)

Ce qui donne ici :

bilbo.initialisation (200,100);

Tout le monde aura compris qu'ici le paramètre *vie* sera égal à 200 et que le paramètre force sera égal à 100.

Ce qui nous donne pour notre classe *Principal*

Ex 2.7 (suite)

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		bilbo.initialisation(200,100);
		}
}
```

Pour vous entrainer, créer un nouveau champ *nom_perso* de type *String* pour notre classe *Personnage*. Modifier la méthode *initialisation* pour qu'elle affiche « Votre personnage se nomme ?????? » (vous remplacerez les ? par le contenu de la variable *nom_perso* !). Créer un nouveau paramètre *nom* (à vous de choisir le bon type) pour la méthode *initialisation* qui permettra au moment de l'appel de cette méthode de faire passer le nom que vous aurez choisi pour votre personnage.

Et voici le résultat :
ex 2.8

classe Personnage :

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public void initialisation (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
}
```

classe Principal :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		bilbo.initialisation(200,100,"Bilbo");
		}
}
```

Surdéfinition (ou surcharge) d'une méthode

En java (et dans tous les langages orientés objets) deux méthodes peuvent porter le même nom à condition qu'elles n'aient pas les mêmes paramètres (ce n'est pas tout à fait vrai, elles peuvent avoir les mêmes paramètres dans certaines situations, on parle alors de redéfinition, nous verrons cela dans le chapitre sur l'héritage). On parle de surdéfinition ou de surcharge d'une méthode.
Prenons tout de suite un exemple pour bien vous faire comprendre l'intérêt de la surcharge d'une méthode.
Imaginons qu'il existe 2 types de personnages: les soldats qui combattent (les forces du mal !) et les civils. Les soldats ont des points de force mais pas les civils. Nous allons donc écrire 2 méthodes *initialisation;* une pour les soldats avec 3 paramètres (*vie* (de type *int*), *force* (de type *int*), *nom* (de type *String*)) et une pour les civils avec uniquement 2 paramètres (*vie* (de type *int*) et *nom* (de type *String*)).

Ex 2.9

Classe Personnage :

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public void initialisation (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage (un
soldat)");
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
	public void initialisation (int vie, String nom){
		point_de_vie = vie;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage (un
civil)");
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
	}

}
```

Classe Principal :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		bilbo.initialisation(200,100,"Bilbo");
		Personnage gollum = new Personnage ();
		gollum.initialisation(150, "Gollum");
		}
}
```

```
Résultat :

Vous venez de créer un nouveau personnage (un soldat)
Il se nomme Bilbo
point de vie : 200
Point de force :100
Vous venez de créer un nouveau personnage (un civil)
Il se nomme Gollum
point de vie : 150
```

Vous avez sans doute remarqué les 2 méthodes *initialisation*, une question se pose :

Comment se fait le choix entre ces 2 méthodes *initialisation* ?

Pour choisir, Java tient compte des paramètres : la version « soldat » de la méthode *initialisation* demande 3 paramètres, alors que la version « civil » d'*initialisation* demande 2 paramètres. Donc si lors de l'appel de la méthode *initialisation* vous passez 3 paramètres, c'est la version « soldat » qui sera appelée, alors que si vous passez 2 paramètres, c'est la version « civil » qui sera appelée.
Tout cela peut être encore plus subtil; nous pouvons avoir le même nombre de paramètres, mais des paramètres de types différents.

Voici un exemple :

Ex 2.10 :

Classe Nombre :

```
public class Nombre {
	public void choix (int x){
		System.out.println("type int");
	}
	public void choix (double x){
		System.out.println("type double");
	}
}
```

Classe Principal :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Nombre nbre = new Nombre ();
		nbre.choix(4);
		nbre.choix(4.0);
		}
}
```

Résultat :

```
type int
type double
```

Cette fois, les 2 versions de la méthode *choix* ont le même nombre de paramètres. Ce qui va différer entre les 2 versions c'est le type du paramètre (un *int* dans un cas et un *double* dans l'autre).
Dans la classe *Principal* nous créons une instance de la classe *Nombre*, puis nous nous servons de cette instance pour appeler la méthode *choix* deux fois : une première fois en lui passant un *int* comme paramètre (4) et une deuxième fois en lui passant un *double* (4.0). Vous pouvez constater que le choix se fait correctement.

champs et méthodes "static"

Jusqu'à présent, tous nos champs étaient des champs d'instance, chaque instance possédait sa propre version de la variable. Il faut savoir qu'il existe aussi des champs de classe : un seul exemplaire de la variable existe pour toutes les instances (chaque instance possède une copie de cette variable), dans certains langages, on parle de membres partagés. Si vous voulez créer un champ de classe, vous devez utiliser le mot clé *static* au moment de l'initialisation du champ.

Exemple : *private static int num_perso* crée un champ de classe *private* de type *int*.
Prenons un exemple pour illustrer tout cela :
Nous allons créer un champ *static* qui nous permettra d'attribuer un numéro à chaque instance nouvellement créée (*num_perso*), le premier personnage créé portera le numéro 1, le deuxième, le numéro 2, etc...
ex 2.11

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	private static int num_perso=1;
	public void initialisation (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("Il porte le numéro " + num_perso);
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
		num_perso=num_perso+1;
	}
}
```

Qu'avons-nous modifié ?
Nous avons ajouté notre nouvelle déclaration (`private static int num_perso;`), il est important de noter que nous avons donné une valeur à notre variable au moment de sa création (ici notre variable vaut 1, alors que par défaut, sa valeur est 0 au moment de sa création). De plus, nous avons ajouté un *println* pour afficher le numéro du personnage qui vient d'être créé.
Enfin, à la dernière ligne, nous avons augmenté la valeur de *num_perso* d'une unité, nous reviendrons plus en détail sur cette ligne dans un prochain chapitre, mais grosso modo, la ligne `"num_perso=num_perso+1"` signifie: « la nouvelle valeur de *num_perso* est égal à l'ancienne valeur de *num_perso* plus un ». Une fois l'initialisation de *bilbo* terminée, la valeur contenue dans le champ *num_perso* sera 2.
N'oubliez pas qu'il n'existe qu'une seule version de la variable, avant la dernière ligne de notre classe, elle valait 1 pour toutes les instances; après cette dernière ligne, elle vaut 2 pour toutes les instances.

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage ();
		bilbo.initialisation(200,100,"Bilbo");
		Personnage gollum = new Personnage ();
		gollum.initialisation (150, 150, "Gollum");
		}
}
```

Nous avons créé 2 instances de la classe Personnage : *bilbo* et *gollum*

Voici le résultat :

```
Vous venez de créer un nouveau personnage
Il porte le numéro 1
Il se nomme Bilbo
point de vie : 200
Point de force :100
Vous venez de créer un nouveau personnage
Il porte le numéro 2
Il se nomme Gollum
point de vie : 150
Point de force :150
```

Voilà, tout fonctionne parfaitement. Notre champ *num_perso static* a été initialisé au moment de la création de l'instance *bilbo*. En revanche, la création de l'instance *gollum* n'a pas entrainé une nouvelle initialisation de notre champ *static num_perso* (sinon, sa valeur aurait été aussi de 1 pour *gollum*) : Un champ *static* est initialisé au moment de la création de la première instance d'une classe. Lors de la création de l'instance *gollum*, on a *num_perso* = 2, d'où le "`Il porte le numéro 2" !`

Pour vous convaincre de l'importance du mot clé *static*, supprimez-le dans la déclaration de notre champ *num_perso*, et notez le résultat :

```
Vous venez de créer un nouveau personnage
Il porte le numéro 1
Il se nomme Bilbo
point de vie : 200
Point de force :100
Vous venez de créer un nouveau personnage
Il porte le numéro 1
Il se nomme Gollum
point de vie : 150
Point de force :150
```

Cela ne fonctionne plus, notre instance *gollum* possède sa propre version du champ *num_perso*, qui est initialisée avec la valeur 1 au moment de la création de notre deuxième instance (*gollum*).

N'hésitez pas à faire vos propres tests pour expérimenter tout cela.

Il faut savoir qu'il existe aussi des méthodes de classe, nous n'allons pas entrer dans les détails, mais vous devez tout de même savoir que :
Une méthode de classe (donc une méthode *static*) sera de la forme
public static void nom_de_la_méthode {
.....
}
(le *void* et le *public* peuvent être remplacés par autre chose, comme pour n'importe quelle méthode)

Une méthode de classe ne peut pas être utilisée avec une instance (on ne pourra pas avoir quelques choses du type : *bilbo.bidule( )* si *bilbo* est une instance et *bidule* est une méthode de classe). Les méthodes de classe devront être utilisées avec des classes : *Personnage.bidule ( )*" si *Personnage* est une classe et *bidule* une méthode de classe.
Voilà, cela devrait vous permettre de vous en sortir avec les méthodes de classe!

Le constructeur

Problème : si la méthode *initialisation* n'est pas appelée dans la méthode *main*, les paramètres ne sont pas transmis, et rien ne s'affiche.
Dans beaucoup de cas en POO (et pas seulement en Java), on est amené à utiliser des classes écrites par d'autres. Imaginez donc que quelqu'un décide d'utiliser notre classe *Personnage* (on peut toujours rêver !!). S'il oublie d'utiliser la méthode *initialisation*, les champs ne seront donc pas initialisés, ceci n'est pas acceptable !
Vous vous doutez bien que les concepteurs de Java ont « prévu le coup », ils ont inventé les constructeurs.

Un constructeur est une méthode qui se lance "automatiquement" à la création d'une instance, elle doit porter le même nom que la classe (ne pas oublier la majuscule). Elle ne renvoie pas de valeurs (malgré cela, le mot clé *void* ne doit pas être utilisé).
Elle accepte des paramètres comme n'importe quelle méthode.
Dans notre exemple, nous allons supprimer la méthode *initialisation* et la remplacer par le constructeur de la classe *Personnage*, c'est-à-dire la méthode *Personnage ()* :

ex 2.12

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public Personnage (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
}
```

Pour la classe *Principal*, le passage des arguments du constructeur devra se faire au moment de la création de l'instance :

ex 2.12 (suite) :

```
public class Principal{

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage (100,200,"Bilbo");
		}
}
```

Plus besoin d'appeler la méthode *initialisation* (qui d'ailleurs n'existe plus dans notre classe *Personnage*)

Pour terminer avec les constructeurs, vous devez savoir qu'ils sont « surchargeables » comme n'importe quelle méthode.

L'encapsulation

Je vais (quand même) vous montrer ce qu'il ne faut jamais faire : rendre les champs *public* (accessibles depuis l'extérieur de la méthode). Nous allons créer une nouvelle méthode : *affiche_etat* (qui comme son nom l'indique permet d'afficher l'état du personnage). De plus, nous allons rendre notre champ *point_de_force public*.

Ex 2.13

```
public class Personnage {
	private int point_de_vie;
	public int point_de_force;
	private String nom_perso;
	public Personnage (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
		System.out.println ("Vous venez de créer un nouveau personnage");
		System.out.println ("Il se nomme " + nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
	public void affiche_etat (){
		System.out.println ("Voici l'état de "+ nom_perso);
		System.out.println ("point de vie : " + point_de_vie);
		System.out.println ("Point de force :" + point_de_force);
	}
}
```

modification de la classe *Principal* :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage (100,200,"Bilbo");
		bilbo.point_de_force = 300;
		System.out.println("on vient de tricher");
		bilbo.affiche_etat();

		}
}
```

```
Résultat :
Vous venez de créer un nouveau personnage
Il se nomme Bilbo
point de vie : 100
Point de force :200
on vient de tricher
Voici l'état de Bilbo
point de vie : 100
Point de force :300
```

Je vous laisse le soin d'analyser tout cela, rien de bien nouveau, cela ne devrait pas vous poser de problème.

En rendant tous vos champs *private* (en interdisant donc l'accès depuis l'extérieur de la classe) vous respectez un des grands principes de la POO : l'encapsulation.
Cette technique permet de garantir que l'objet sera correctement utilisé.

accesseurs et mutateurs

Si vous rétablissez le caractère *private* de notre champ *point_de_force*, Eclipse va vous sortir une splendide erreur, car non seulement vous ne pouvez plus modifier ce champ depuis la méthode *main*, mais vous ne pouvez même plus lire ces champs depuis « l'extérieur » de la classe *Personnage*. Pour vous en convaincre, essayez de faire fonctionner le programme suivant :

Ex 2.14

modification de la classe *Personnage* :

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public Personnage (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
	}
}
```

Vous pouvez constater que nous avons effacé une partie du constructeur (la partie qui affichait le nom et le nombre de points)

modification de la classe *Principal* :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage (100,200,"Bilbo");
		System.out.println ("Votre personnage se nomme "+ bilbo.nom_perso);

		}
}
```

Nous essayons d'afficher le champ *nom_perso* de l'instance *bilbo* (*bilbo.nom_perso*), mais avant même de lancer le programme, Eclipse nous affiche une belle croix rouge qui signifie : "Je suis désolé, mais le champ *nom_perso* n'est pas accessible depuis la classe *Principal*)

Vous allez me dire « Mais pourquoi avez-vous modifié le constructeur, cela fonctionnait très bien avant ? »
C'est vrai, mais si au cours du « jeu » il vous prend l'envie de consulter ces informations, comment allez-vous faire ? (ils auront été affiché au moment de la création de l'instance, mais après, terminé, les informations deviennent inaccessibles)
Vous vous doutez bien qu'il existe une solution : les accesseurs et les mutateurs (setter et getter en Anglais)

Un mutateur est une méthode qui va permettre de modifier un champ privé depuis l'extérieur de la classe où il a été créé.
Un accesseur est une méthode qui va permettre de lire un champ privé depuis l'extérieur de la classe où il a été créé.
Vous allez encore me dire : "Quel est l'intérêt de rendre les champs privés, si on crée des méthodes capables de les modifier ?"
Très bonne question, les mutateurs et les accesseurs pourront être rendus utilisables seulement sous certaines conditions, conditions qui auront été définies par le créateur de la classe. Alors que si les champs sont *public*, tout le monde pourra faire n'importe quoi, sans aucun contrôle.

Commençons par écrire les accesseurs pour notre classe *Personnage* :
Par convention, le nom des accesseurs devra commencer par *get*, on aura *getNomDuChamp.*
Nous allons donc créer trois méthodes dans la classe *Personnage* : *getPointDeVie*, *getPointDeForce* et *getNomPerso*. La méthode *getPointDeVie* renverra un entier (*int*) : la valeur du champ *point_de_vie*. La méthode *getPointDeForce* renverra aussi un entier : la valeur du champ *point_de_force*. La méthode *getNomPerso* renverra une chaine (*String*) : la chaine contenue dans le champ *nom_perso*. L'écriture de ces méthodes ne pose pas de problème (on utilise le mot clé *return*, comme dans Alice) :

Ex 2.15

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public Personnage (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
	}
	public int getPointDeVie(){
		return point_de_vie;
	}
	public int getPointDeForce (){
		return point_de_force;
	}
	public String getNomPerso (){
		return nom_perso;
	}
}
```

Nous pouvons maintenant modifier notre classe *Principal*

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage (100,200,"Bilbo");
		System.out.println ("Nom : "+ bilbo.getNomPerso());
		System.out.println ("point de vie "+ bilbo.getPointDeVie());
		System.out.println ("point de force "+ bilbo.getPointDeForce());
		}
}
```

Voici le résultat :

```
Nom : Bilbo
point de vie 100
points de force 200
```

Les champs sont donc bien accessibles (indirectement) depuis la classe *Principal*

Écrivons maintenant les mutateurs pour notre classe *Personnage* :

Toujours par convention, les mutateurs commencent toujours par *set*. On aura donc ici *setPointDeVie*, *setPointDeForce* et *setNomPerso*. Les mutateurs permettant de modifier la valeur des champs, on devra utiliser des paramètres (nos mutateurs ressembleront beaucoup à la méthode *initialisation*).

Voici notre classe *Personnage* avec ses mutateurs

Ex 2.16

```
public class Personnage {
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;
	public Personnage (int vie, int force, String nom){
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
	}
	public int getPointDeVie(){
		return point_de_vie;
	}
	public int getPointDeForce (){
		return point_de_force;
	}
	public String getNomPerso (){
		return nom_perso;
	}
	public void setPointDeVie (int vie){
		point_de_vie = vie;
	}
	public void setPointDeForce (int force){
		point_de_force = force;
	}
	public void setNomPerso (String nom){
		nom_perso = nom;
	}
}
```

Pour utiliser nos mutateurs, modifions la classe *Principal* :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		Personnage bilbo = new Personnage (100,200,"Bilbo");
		System.out.println ("Nom : "+ bilbo.getNomPerso());
		System.out.println ("point de vie "+ bilbo.getPointDeVie());
		System.out.println ("point de force "+ bilbo.getPointDeForce());
		bilbo.setPointDeForce(300);
		System.out.println ("point de force modifié");
		System.out.println ("point de force "+ bilbo.getPointDeForce());


		}
}
```

Résultat :

```
Nom : Bilbo
point de vie 100
point de force 200
point de force modifiés
points de force 300
```

Juste une remarque pour terminer, il ne faut pas confondre le nom de l'instance (*bilbo* par exemple) et le champ *nom_perso*. Grâce à notre mutateur, nous pouvons modifier le champ *nom_perso* de l'instance sans toucher au nom de l'instance.

Chapitre 3
Commentaires et première fenêtre

Les commentaires

Nos programmes commencent à être de plus en plus compliqués, et cela ne va pas aller en s'arrangeant !
Pour qu'ils restent malgré tout le plus clair possible, nous allons systématiquement ajouter des commentaires (nous avons déjà entrevu cette possibilité avec Alice)
Un programme non commenté devient très rapidement illisible, même pour un programmeur expérimenté.
Voici un exemple de programme commenté :

Ex 3.1

```
//Début de la classe personnage
public class Personnage {

	//Mise en place des champs
	private int point_de_vie;
	private int point_de_force;
	private String nom_perso;

	//Constructeur de la classe Personnage
	public Personnage (int vie, int force, String nom){
		//Passage des paramètres aux champs d'instance
		point_de_vie = vie;
		point_de_force = force;
		nom_perso = nom;
	}

	//Accesseur point_de_vie
	public int getPointDeVie(){
		return point_de_vie;
	}

	//Accesseur point_de_force
	public int getPointDeForce (){
		return point_de_force;
	}

	//Accesseur nom_perso
	public String getNomPerso (){
		return nom_perso;
	}

	//Mutateur point_de_vie
	public void setPointDeVie (int vie){
		point_de_vie = vie;
	}

	//Mutateur point_de_force
	public void setPointDeForce (int force){
		point_de_force = force;
	}

	//Mutateur nom_perso
	public void setNomPerso (String nom){
		nom_perso = nom;
	}
}
```

Vous avez sans doute remarqué que les commentaires commencent par //. Il ne faut pas hésiter à "aérer" votre programme (saut de ligne).
Même si cela va vous paraitre contraignant au début, n'hésitez donc pas à commenter vos programmes le plus clairement possible, de manière concise et informative
NB : Dans l'exemple 3.1, les commentaires ne sont pas très « intéressants », normalement, il faut éviter les « Lapalissades » !!

Nos premières fenêtres et notion de variable locale

Jusqu'à présent nos programmes étaient en "mode console", nous allons écrire nos premiers programmes en mode "graphique" (avec les fenêtres, les boutons,....). Nous étudierons en détail ce mode « graphique » dans un prochain chapitre, mais par souci de confort, nous allons « construire » nos premières fenêtres dès maintenant.

« Construire » n'est pas vraiment le terme approprié. En effet les classes permettant d'afficher les fenêtres existent déjà et nous allons bien évidemment les utiliser. Vous allez enfin pouvoir « admirer » toute la puissance de la programmation orientée objet avec ses classes « réutilisables» !

Nous allons donc (pour la première fois !) utiliser des classes « toutes faites », mais avant de pouvoir les utiliser, nous allons devoir les importer dans notre projet grâce au mot clé *import*
Il existe beaucoup de classes pré-existantes, elles sont donc regroupées dans des *packages*. Le package qui contient les classes permettant d'afficher des fenêtres s'appelle *swing* (il y en a d'autres, mais nous, nous utiliserons *swing*). Il existe 2 types de packages, les packages *java* et les packages *javax* (je ne rentrerai pas dans les détails !), notre package est de type *javax*.

Pour importer notre classe, nous utiliserons la ligne suivante :

import javax.swing.JOptionPane ;

Je pense que vous avez tous compris que la classe que nous allons utiliser pour afficher nos fenêtres se nomme *JOptionPane* !

Pour afficher une fenêtre à l'écran nous allons utiliser une méthode *static* (méthode de classe) de la classe *JOptionPane* : *showMessageDialog*

Cette méthode *showMessageDialog*, requiert 2 arguments. On aura donc une ligne du type :

JOptionPane.showMessageDialog (argument 1, argument 2)

avec *null* à la place d'*arguments 1* et la chaine de caractère que vous voulez afficher dans votre fenêtre à la place d'*argument 2*

Passons à un exemple :
Ex 3.2

```
//Nous importons la classe JOptionPane
import javax.swing.JOptionPane;
public class Fenetre {
	// Pas de paramètres dans cet exemple
	public static void main(String[] args) {
		//Notre première fenêtre
		JOptionPane.showMessageDialog(null, "Bonjour le monde !");
	}

}
```

Résultat :

Pour faire disparaitre la fenêtre (et donc arrêter le programme) il suffit de cliquer sur OK ou sur la croix rouge.
Vous pouvez déplacer cette fenêtre sans problème, tout cela est geré automatiquement par la classe *JoptionPane*.
Admirez toute la puissance de la POO : vous ne savez pas du tout comment fonctionne la classe *JOptionPane* et cela n'a aucune importance. Vous devez uniquement apprendre à utiliser les méthodes de cette classe (paramètres).

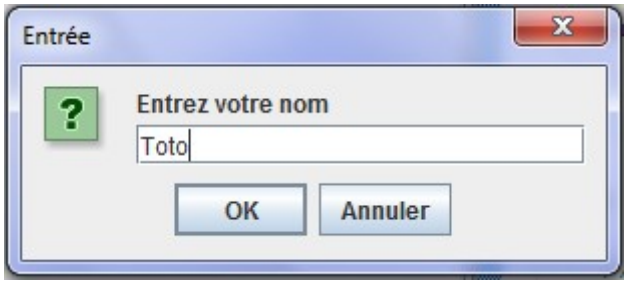
Nous allons maintenant voir un deuxième type de fenêtre.
Jusqu'à présent, nos programmes ne sont pas très interactifs (l'utilisateur n'a strictement rien à faire, juste à regarder le résultat). Avec la méthode *showInputDialog*, l'utilisateur va pouvoir entrer une chaine de caractères au clavier. Cette chaine de caractère sera stockée dans une variable, cette variable pourra être utilisée ultérieurement.
Voici un exemple :
Ex 3.3

```
//Nous importons la classe JOptionPane
import javax.swing.JOptionPane;
public class Fenetre {
	// Pas de champ d'instance dans cet exemple

	//Voici notre méthode main
	public static void main(String[] args) {
		//déclaration d'une variable locale
		String nom ;
		//fenêtre qui permet à l'utilisateur d'entrer son nom
		nom = JOptionPane.showInputDialog("Entrez votre nom");
		//fenêtre qui affiche "Bonjour nom utilisateur"
		JOptionPane.showMessageDialog(null, "Bonjour "+nom);
		}

}
```

Résultat :



Alors, si nous prenons ce programme ligne par ligne, la première chose qui attire notre attention, c'est le commentaire à la ligne 7 : `//déclaration d'une variable locale`
variable locale ??

Nous avons déjà vu cette notion de variable locale dans Alice (voir le dernier chapitre de "Apprendre la POO avec Alice").

La variable de type *String nom* est déclarée dans une méthode (ici la méthode *main*), c'est une variable locale.
Les variables *point_de_vie*, *point_de_force*,..... (voir les exemples du chapitre 2), sont déclarées en dehors de toutes méthodes (comme nous l'avons déjà vu, se sont donc des champs d'instances (ou des champs de classe si les variables sont déclarées avec le mot clé *static*)).
Les variables locales n'existent pas en dehors des méthodes où elles ont été déclarées (d'où le nom de locales). Dans notre programme, dès que la méthode *main* se termine, la variable locale *nom* est détruite (elle disparait de la mémoire).

Pour illustrer notre propos, nous allons réécrire un programme avec 2 classes (il donnera exactement le même résultat que l'exemple 3.3

Ex 3.4

Classe *Fenetre* :

```
//Nous importons la classe JOptionPane
import javax.swing.JOptionPane;
public class Fenetre {
	//Création du champ d'instance nom
	private String nom;
	// Création de la méthode qui demande le nom
	public void fen_demande_nom (){
		nom = JOptionPane.showInputDialog("Entrez votre nom");
	}
	// Création méthode qui affiche le nom
	public void fen_affiche_nom (){
		JOptionPane.showMessageDialog(null, "Bonjour "+nom);
	}
}
```

Classe *Principal* (avec la méthode *main*) :

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		//Création d'une instance de la classe Fenetre
		Fenetre fen1=new Fenetre ();
		fen1.fen_demande_nom();
		fen1.fen_affiche_nom();
	}

}
```

Je vous laisse étudier ce programme, cela ne devrait pas poser de problème.

Modifier la classe *Fenetre* (la variable *nom* perd son « statut » de champ d'instance et devient une variable locale) :

Ex 3.5

Classe *Fenetre*

```
import javax.swing.JOptionPane;
public class Fenetre {
	public void fen_demande_nom (){
		// variable locale nom
		String nom ;
		nom = JOptionPane.showInputDialog("Entrez votre nom");
	}
	public void fen_affiche_nom (){
		JOptionPane.showMessageDialog(null, "Bonjour "+nom);
	}
}
```

La classe *Principal* reste inchangée

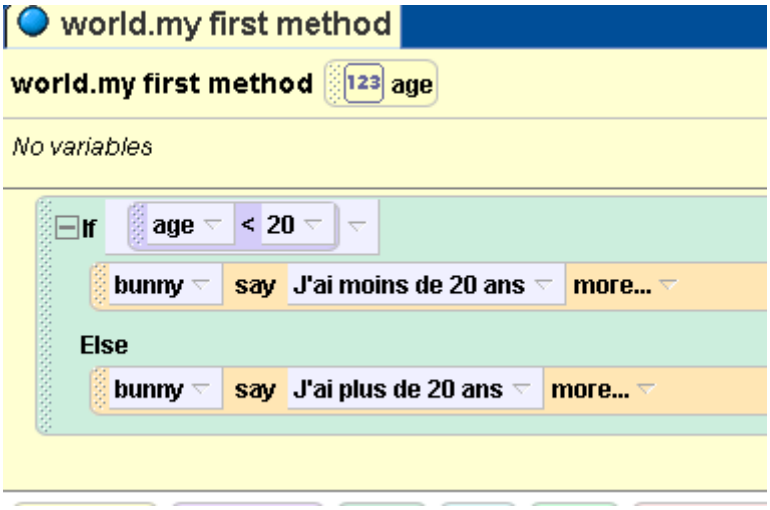
Le programme ne fonctionne plus. Dans un premier temps essayez de trouver pourquoi par vous même !
Vous avez trouvé ?

La variable *nom* est une variable locale, elle est donc « détruite » dès que le programme sort de la méthode *fen_demande_nom*. Or, la variable *nom* est utilisée dans la méthode *fen_affiche_nom*. Cela ne peut donc pas fonctionner !!

Chapitre 4
Les structures de contrôle

if / else

Nous avons, à de nombreuses reprises, utilisé le couple if/else dans Alice, vous ne devriez donc pas être trop dépaysés !



L'exemple ci-dessus ne devrait vous poser aucun problème (dans le cas contraire, n'hésitez pas à revoir le cours sur Alice) : si la variable *age* est inférieur à 20 alors le lapin dit : « J'ai moins de 20 ans », si la variable *age* est supérieur ou égale à 20 alors le lapin dit : « J'ai plus de 20 ans ».

Je vous rappelle tout de même que la structure générale est du type :

```
if (expression) {
        instruction 1 ;
}
else {
        instruction 2 ;
}
instruction 3 ;
```

« expression » doit forcement renvoyer un booléen (revoir le cours sur Alice). Si « expression » renvoie *true* (vrai) l'*instruction 1* est exécutée (mais pas *instruction 2*), si « expression » renvoie *false* (faux) *instruction 2* est exécutée (mais pas *instruction 1*). Dans les 2 cas, *instruction 3* est exécutée (nous sommes « sortis » du if/else).

J'ai volontairement respecté la syntaxe de java dans l'exemple que je viens de vous donner. Vous pouvez donc constater que tout cela n'est pas très difficile à mettre en œuvre. Passons donc à notre premier exemple en java :

Ex 4.1

```
//importation de la classe JOptionPane
import javax.swing.JOptionPane;
public class Principal {
	public static void main(String[] args) {
		//Création de 2 variables locales
		String age_S;
		int age_i;
		//affichage de la fenêtre
		age_S = JOptionPane.showInputDialog("Quel est votre age ?");
		//transformation de la chaine en int
		age_i = Integer.parseInt(age_S);
		if (age_i<20){
			JOptionPane.showMessageDialog(null, "Vous avez moins de 20
ans");
		}
		else{
			JOptionPane.showMessageDialog(null, "Vous avez plus de 20
ans");
		}
	}

}
```

Dans ce programme, nous avons utilisé 2 variables locales : *age_S* de type *String* et *age_i* de type *int*. Pourquoi ?

Cela ne vous parait peut-être pas évident, mais nous avons déjà eu ce genre de problème avec Alice (voir la programmation du chrono dans le dernier chapitre). Quand vous entrez votre âge dans la fenêtre appropriée, vous tapez une chaine de caractères, voilà pourquoi *age_S* est de type *String*. Essayez de remplacer age_S par *age_i* (de type *int*) et vous aurez droit à une belle erreur (*type mismatch*) ! Bref, la variable qui « récupère » ce qui est tapé au clavier est forcément de type String.

Problème : un peu plus bas, au niveau du if, nous avons une comparaison de 2 entiers (nous ne pouvons pas comparer un entier (20) avec une chaine de caractères (*age_S*)) d'où l'existence de la variable *age_i* de type *int* et de la « transformation » de la chaine en *int* grâce à la ligne :

age_i = Integer.*parseInt*(age_S) ;

parseInt est une méthode qui « transforme » une chaine en *int* (c'est une méthode de la classe *Integer*).
La valeur renvoyée par la méthode *parseInt* est ensuite affectée à la variable *age_i*.
Vous aurez sans aucun doute à utiliser ce "tour de passe-passe" plusieurs fois dans vos futurs programmes.

La suite du programme ne devrait pas vous poser de problème, c'est un exemple ultra classique de l'utilisation du couple if / else :

Si *age_i* est inférieur à vingt, l'expression (*age_i < 20*) renvoie *true* l'instruction qui est juste en-dessous du if est exécutée.
Si *age_i* est supérieur à vingt, l'expression (*age_i < 20*) renvoie *false* et c'est alors ce qui suit le *else* qui est exécuté.

Et si *age_i* est égal à vingt, que se passe-t-il ? A vous de me le dire !

Il existe une autre structure intéressante :

if

else if

else

Nous allons l'utiliser dans l'exemple suivant :

Ex 4.2

```
//importation de la classe JOptionPane
import javax.swing.JOptionPane;
public class Principal {
	public static void main(String[] args) {
		//Création de 2 variables locales
		String age_S;
		int age_i;
		//affichage de la fenêtre
		age_S = JOptionPane.showInputDialog("Quel est votre age ?");
		//transformation de la chaine en int
		age_i = Integer.parseInt(age_S);
		if (age_i<20){
			JOptionPane.showMessageDialog(null, "Vous avez moins de 20
ans");
		}
		else if (age_i>20){
			JOptionPane.showMessageDialog(null, "Vous avez plus de 20
ans");
		}
		else {
			JOptionPane.showMessageDialog(null, "Vous avez 20 ans");
		}
	}

}
```

Vous pouvez enchainer les *else if*.

Cette structure n'est pas très compliquée à comprendre, je vous laisse réfléchir à tout cela.

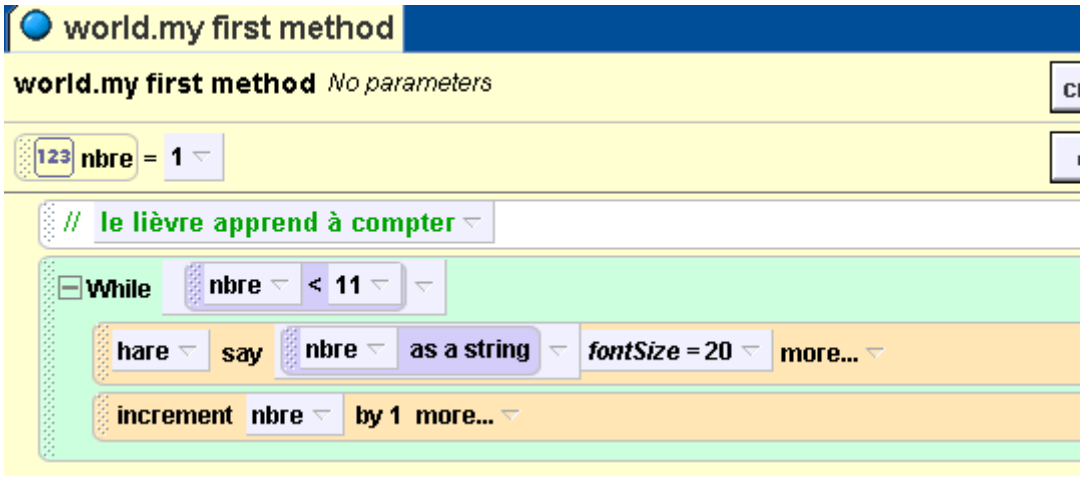
Nous avons utilisé 2 opérateurs de comparaison "supérieur à" et "inférieur à".Il en existe d'autres, voici un petit tableau pour résumer :

Opérateur	Opération réalisée	Exemple	Résultat de l'exemple
= =	Egalité	4 = = 5	false
! =	Inégalité	4 ! = 5	true
<	Inférieur	2 < 5	true
>	Supérieur	2 > 5	false
< =	Inférieur ou égal	2 < = 5	true
> =	Supérieur ou égal	2 > = 5	false

J'attire tout particulièrement votre attention sur l'opérateur égalité : = =
Il ne faut pas confondre cet opérateur avec le simple signe égal = qui permet d'attribuer une valeur à une variable (sauf pour les variables de type String, mais là, c'est une autre histoire); soyez donc très attentifs à cela, sinon, vous aurez le droit à une belle erreur de syntaxe.

While

Dans Alice nous avons déjà eu l'occasion d'utiliser l'instruction *while* à plusieurs reprises.
Voici un exemple pour vous rafraichir la mémoire (le lièvre compte jusqu'à 10)



while signifie « tant que ». Dans l'exemple « Alice » (ci-dessus), nous avons créé une variable locale *nbre*. Tant que *nbre* reste inférieur à 11, le lièvre affiche la valeur de *nbre*, à la fin du bloc *while*, la variable *nbre* est augmentée d'une unité.

Voici l'équivalent Java de ce programme :

Ex 4.3

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		//Création de notre variable locale
		int nbre = 1;
		while (nbre<11){
			System.out.println (nbre);
			nbre=nbre+1;

		}

	}

}
```

Rien de très difficile dans cet exemple, vous pouvez constater que l'utilisation du *while* en Java est assez simple.

Il faut tout de même savoir que la ligne *nbre=nbre + 1* peut être remplacée par *nbre ++* (c'est un raccourci)

On a donc :

Ex 4.3 (bis)

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		//Création de notre variable locale
		int nbre = 1;
		while (nbre<11){
			System.out.println (nbre);
			nbre++;

		}

	}

}
```

Voici le résultat :

```
1
2
3
4
5
6
7
8
9
10
```

Nous allons maintenant voir des exemples qui vont nous permettre de travailler quelques algorythmes importants.

Répétition contrôlée par compteur :

Voici un programme qui permet de calculer la moyenne de 10 élèves (la note doit être un entier). Je vous laisse analyser tout seul ce programme : concentrez-vous sur la boucle while car c'est un algorythme classique (répétition contrôlée par compteur).

Ex 4.4

```
import javax.swing.JOptionPane;

public class Moyenne {

	public static void main(String[] args) {
		//variables locales
		int total;
		int compteur;
		int note_i;
		String note_S;
		int moyenne;
		// initialisation des variables
		total=0;
		compteur=1;
		// boucle (10 fois)
		while (compteur <=10){
			//fenêtre pour entrer une note
			note_S=JOptionPane.showInputDialog("Entrez une note entière");
			//convertir String en int
			note_i=Integer.parseInt(note_S);
			//ajoute la dernière note entrée au total
			total=total+note_i;
			//incremente le compteur d'une unité
			compteur++;
		}
		//calcul de la moyenne
		moyenne=total/10;
		//affichage de la moyenne
		JOptionPane.showMessageDialog(null, "Moyenne = "+moyenne);
		//fin programme
		System.exit(0);
	} // fin méthode main
} //fin classe moyenne
```

Ce programme a un gros défaut : l'obligation d'entrer 10 notes (ni plus ni moins).
Nous allons réécrire le même programme à une exception prêt, le nombre de notes à entrer ne sera pas écrit en « dur » dans le programme; pour mettre fin à la saisie, l'utilisateur devra entrer *-1* à la place d'une note.
Pour se faire, nous allons voir un autre algorithme important : « la répétition contrôlée par sentinelle » (le programme « surveille » la fin de la saisie, d'où le terme de sentinelle).

Ex 4.5

```
import javax.swing.JOptionPane;

public class Moyenne {

	public static void main(String[] args) {
		//variables locales
		int total;
		int compteur;
		int note_i;
		String note_S;
		int moyenne;
		// initialisation des variables
		total=0;
		compteur=0;
		//fenêtre pour entrer la première note
		note_S=JOptionPane.showInputDialog("Entrez une note entière");
		//convertir String en int pour la première note
		note_i=Integer.parseInt(note_S);
		// début boucle controlée par sentinelle
		while (note_i != -1){
			//ajoute la dernière note entrée au total
			total=total+note_i;
			//incremente le compteur d'une unité
			compteur++;
			//fenêtre pour entrer une note
			note_S=JOptionPane.showInputDialog("Entrez une note entière");
			//convertir String en int
			note_i=Integer.parseInt(note_S);
		}
		//calcul de la moyenne
		moyenne=total/compteur;
		//affichage de la moyenne
		JOptionPane.showMessageDialog(null, "Moyenne = "+moyenne);
		//fin programme
		System.exit(0);
	} // fin méthode main
} //fin classe moyenne
```

Je vous laisse analyser ce programme seul, en vous aidant de quelques petites remarques :

- Au niveau de la déclaration des variables, rien de nouveau, en revanche, la variable *compteur* est initialisée avec la valeur zéro, essayez de comprendre pourquoi
- La saisie de la 1ère note se fait en dehors de la boucle *while*
- Nous avons une inversion de lignes dans la boucle

Bref je vous conseille de bien travailler cet exemple, c'est un grand classique !

Notre programme n'est pas encore « parfait », nous allons encore le modifier.

La variable moyenne est de type *int*, le résultat qui s'affiche est donc un entier. Or, si nous faisons les calculs « à la main », dans la plupart des cas, le résultat est un nombre à virgule. Nous allons modifier notre programme pour qu'il affiche comme résultat, un nombre à virgule.

Vous allez me dire « c'est simple, il suffit que la variable moyenne soit de type double ! »
Je vous répondrai : « Oui bien sûr, mais essayez l'exemple suivant (à la calculatrice et avec votre programme) »

Notes : 12 ; 14 ; 15 => à la calculatrice : 12,7 avec le programme : 13,0
Cela ne fonctionne pas (bien), mais d'où vient le problème ?

Dans l'opération *moyenne = total / compteur*, *total* est de type *int*, tout comme *compteur*. Or, un *int* divisé par un "int" donne un "int" ! (enfin pas tout à fait puisque l'on a "13,0", mais vous n'aurez jamais un résultat du type "12.7").
Nous allons devoir faire un transtypage :
moyenne = (double) total / compteur
Sans trop entrer dans les détails, Java crée une copie de la variable *total* de type *double* et s'en sert pour effectuer l'opération demandée (il « transforme » aussi *compteur* en *double* car Java ne peut évaluer des expressions arithmétiques que lorsqu'elles ne contiennent que des opérandes (nombres qui « participent » à l'opération) de types de données identiques, on dit que Java a promu *compteur* de *int* à *double*.)

Ex 4.5 (bis)

```
import javax.swing.JOptionPane;

public class Moyenne {

	public static void main(String[] args) {
		//variables locales
		int total;
		int compteur;
		int note_i;
		String note_S;
		double moyenne;
		// initialisation des variables
		total=0;
		compteur=0;
		//fenêtre pour entrer la première note
		note_S=JOptionPane.showInputDialog("Entrez une note entière");
		//convertir String en int la première note
		note_i=Integer.parseInt(note_S);
		// début boucle controlée par sentinelle
		while (note_i != -1){
			//ajoute la dernière note entrée au total
			total=total+note_i;
			//incremente le compteur d'une unité
			compteur++;
			//fenêtre pour entrer une note
			note_S=JOptionPane.showInputDialog("Entrez une note entière");
			//convertir String en int
			note_i=Integer.parseInt(note_S);
		}
		//calcul de la moyenne
		moyenne=(double)total/compteur;
		//affichage de la moyenne
		JOptionPane.showMessageDialog(null, "Moyenne = "+moyenne);
		//fin programme
		System.exit(0);
	} // fin méthode main
} //fin classe moyenne
```

Pour terminer, un exercice à faire par vous même.
Avez-vous essayé de taper *-1* dès la première note ?

Nous avons un drôle de résultat : *Moyenne* = *NaN*, ce qui veut dire « opération mathématiques non admise »
Essayez de comprendre pourquoi et trouvez un moyen pour que ce message un peu étrange soit remplacé par une belle fenêtre avec « Vous devez entrer au moins une note ! » écrit dedans. Allez, je vous aide un peu, cherchez du côté du couple if /else.

Réponse :

```
import javax.swing.JOptionPane;

public class Moyenne {

	public static void main(String[] args) {
		//variables locales
		int total;
		int compteur;
		int note_i;
		String note_S;
		double moyenne;
		// initialisation des variables
		total=0;
		compteur=0;
		//fenêtre pour entrer la première note
		note_S=JOptionPane.showInputDialog("Entrez une note entière");
		//convertir String en int la première note
		note_i=Integer.parseInt(note_S);
		// début boucle controlée par sentinelle
		while (note_i != -1){
			//ajoute la dernière note entrée au total
			total=total+note_i;
			//incremente le compteur d'une unité
			compteur++;
			//fenêtre pour entrer une note
			note_S=JOptionPane.showInputDialog("Entrez une note entière");
			//convertir String en int
			note_i=Integer.parseInt(note_S);
		}
		if (compteur !=0){
			//calcul de la moyenne
			moyenne=(double)total/compteur;
			//affichage de la moyenne
			JOptionPane.showMessageDialog(null, "Moyenne = "+moyenne);
		}
		else {
			//affichage erreur
			JOptionPane.showMessageDialog(null, "Vous devez entrer au
moins une note");
		}
		//fin programme
		System.exit(0);
	} // fin méthode main
} //fin classe moyenne
```

Voilà, nous avons terminé avec les structures de contrôles vues dans Alice, mais Java en posséde d'autres.

La structure de répétition for

Lorsque vous connaissez le nombre d'itérations à réaliser dans une boucle (nombre de fois que la boucle s'exécutera), il est préférable d'utiliser la structure for.

for (initialisation ; condition ; instruction d'itération)
{
    instructions
    .....
}

voici un exemple

Ex 4.7

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		for (int i=0; i<=10; i++)
		{
			System.out.println(i);
		}
	}
}
```

La structure de décision switch

La structure switch permet un fonctionnement équivalent à la structure "if / if else / ... / else" (vue au début du chapitre).

Nous avons une structure du type :

Switch (expression)
{
  Case valeur1 :
      instructions1
     .....
     break ;
  Case valeur2 :
     instructions2
    .....
    break ;
  Case valeur3 :
    instructions3
    .....
    break ;
  Default :
    instructions4
   .....
}

« expression » est souvent une variable. valeur1, valeur2, ... sont les valeurs possibles de cette variable. Si « expression » est égale à valeur1, c'est instructions1..... qui vont être exécutées (jusqu'à l'instruction *break*), si « expression » est égale à valeur2, c'est «instructions2..... qui vont être exécutées (jusqu'à l'instruction *break*) et ainsi de suite.

Voici un exemple d'abord traité avec des *if / if else / .... / else* puis ensuite avec un *switch*

Ex 4.8

```
import javax.swing.JOptionPane;

public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		String i_S;
		int i_int;
		i_S=JOptionPane.showInputDialog("Entrez un chiffre entre 1 et 5 :");
		i_int=Integer.parseInt(i_S);
		if (i_int == 1){
			JOptionPane.showMessageDialog(null, "Vous avez tapé un");
		}
		else if (i_int == 2){
			JOptionPane.showMessageDialog(null, "Vous avez tapé deux");
		}
		else if (i_int == 3){
			JOptionPane.showMessageDialog(null, "Vous avez tapé trois");
		}
		else if (i_int == 4){
			JOptionPane.showMessageDialog(null, "Vous avez tapé quatre");
		}
		else if (i_int == 5){
			JOptionPane.showMessageDialog(null, "Vous avez tapé cinq");
		}
		else {
			JOptionPane.showMessageDialog(null, "Vous n'avez pas respecté
les consignes");
		}

	}
}
```

Ex 4.8bis

```
import javax.swing.JOptionPane;

public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		String i_S;
		int i_int;
		i_S=JOptionPane.showInputDialog("Entrez un chiffre entre 1 et 5 :");
		i_int=Integer.parseInt(i_S);
		switch (i_int)
		{
		case 1 :
			JOptionPane.showMessageDialog(null, "Vous avez tapé un");
			break ;
		case 2 :
			JOptionPane.showMessageDialog(null, "Vous avez tapé deux");
			break ;
		case 3 :
			JOptionPane.showMessageDialog(null, "Vous avez tapé trois");
		case 4 :
			JOptionPane.showMessageDialog(null, "Vous avez tapé quatre");
			break;
		case 5 :
			JOptionPane.showMessageDialog(null, "Vous avez tapé cinq");
		default :
			JOptionPane.showMessageDialog(null, "Vous n'avez pas respecté
les consignes");
		}

	}
}
```

La valeur à tester peut être contenue dans une variable (comme ci-dessus), mais elle peut également être le résultat d'un calcul. Attention, le type de la valeur testée peut-être numérique entier ou un caractère (type *char*).
Utilisation avec le type char :

```
char i ;
switch (i)
{
    case 'o' :
       System.out.println ("Réponse positive") ;
       break ;
    case 'n' :
       System.out.println ("Réponse négative") ;
       break ;
    default :
       System.out.println ("Pas de réponse") ;
}
```

J'attire votre attention sur l'utilisation du ' (alors que pour une chaine on utilise ", **mais attention pas de chaine (*String*) avec un switch**)

Chapitre 5
Notion d'héritage et de polymorphisme

Les notions d'héritage et de polymorphisme sont assez compliquées à appréhender. Mais, ce sont des notions « piliers » de la POO, nous allons donc les aborder, sans trop entrer dans les détails, le but étant ici est de comprendre l'essentiel de ces concepts pour pouvoir poursuivre notre étude du Java (notamment la programmation graphique).

Vous avez déjà eu un premier contact avec la notion d'héritage. Souvenez-vous, dans Alice, la patineuse qui ne savait pas patiner, cela vous dit quelque chose ? Si nécessaire, rendez-vous p 53 du document « Apprendre la POO avec Alice ».

Pour vous résumer la situation, dans Alice la classe *IceSkater* ne possède pas de méthode *patiner*, nous en avons donc écrit une. Pour ne pas être obligé de réécrire cette méthode à chaque fois, nous avons créé une nouvelle classe *SuperIceSkater* qui non seulement possède notre nouvelle méthode *patiner* mais aussi toutes les méthodes de *IceSkater*. On dit alors que la classe *SuperIceSkater* hérite de la classe *IceSkater* (elle a hérité des méthodes mais aussi des champs).

Voyons maintenant comment cela se passe en Java avec un exemple très simple.

Nous allons écrire 2 classes. La première classe, la classe *Personnage*, possède 3 méthodes (pour simplifier les choses, dans un premier temps, nous allons écrire des classes sans champ et sans constructeur), la méthode *marcher*, la méthode *manger* et la méthode *dormir*.

Ex 5.1 :

```
public class Personnage {
	public void marcher (){
		System.out.println("Je marche");
	}
	public void dormir (){
		System.out.println("Je dors");
	}
	public void manger (){
		System.out.println("Je mange");
	}
}
```

Difficile de faire plus simple, non ?

Parmi nos personnages, nous allons avoir des chevaliers, des archers, des magiciens.....

Prenons le cas du magicien : un magicien fait de la magie, mais comme tous les personnages, il peut aussi manger, dormir et marcher.

Nous allons donc écrire une classe *Magicien* avec les méthodes suivantes : *faire_de_la_magie*, *marcher*, *dormir* et *manger*

Les 3 dernières méthodes seront communes à tous les personnages, devrons-nous nous donner la peine de les réécrire à chaque fois ? Comme déjà dit précédemment, la puissance de la POO est liée à sa capacité à réutiliser des choses existantes (éviter de « réinventer la roue à chaque fois »), donc, la réponse à ma question est bien évidemment NON, nous n'allons pas récrire toutes les méthodes à chaque fois. Nous allons utiliser l'héritage : la classe *Magicien* héritera de la classe *Personnage*.

Ex 5.1 (suite)

```
public class Magicien extends Personnage {
	public void faire_de_la_magie () {
		System.out.println ("Je fais de la magie");
	}
}
```

Voici notre classe *Magicien*, elle possède la méthode *faire_de_la_magie* mais aussi toutes les méthodes de la classe *Personnage*. Pour vous prouver cela, écrivons une nouvelle classe avec une méthode main.

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// créons une instance de Magicien
		Magicien gandalf = new Magicien ();
		// ulisation de la méthode faire_de_la_magie
		gandalf.faire_de_la_magie();
		// utilisation de la méthode héritée dormir
		gandalf.dormir();
		}
}
```

Voilà, tout fonctionne Gandalf peut faire de la magie mais aussi manger, dormir et marcher.

Magicien est une classe dérivée de la classe *Personnage*. Nous pouvons aussi dire que *Personnage* est la classe de base de *Magicien*

Dans l'avenir, nous écrirons de nouvelles classes qui hériteront de classes pré-existantes.

Attention
Une méthode d'une classe dérivée n'a pas accès aux membres (champs et méthodes) privés de sa classe de base. Cela veut donc dire que les champs privés de la classe *Personnage* ne seront accessibles depuis la classe *Magicien* (cela, bien sûr, pour respecter l'encapsulation). Pour avoir accès à ces champs, il faudra passer par les accesseurs et les mutateurs de la classe *Personnage* (les accesseurs et les mutateurs sont des méthodes comme les autres, la classe *Magicien* en héritera et pourra donc les utiliser).

La redéfinition d'une méthode et le mot clé « super »

Nous allons maintenant « étoffer » un peu notre classe *Personnage* en lui ajoutant des champs d'instance (*point_de_vie* et *point_de_force*) et une méthode *affichage*. Nous allons aussi ajouter une méthode *affichage_magicien* à notre classe *Magicien*, ainsi qu'un champ d'instance (*point_de_magie*).

Voici nos nouvelles classes :

Ex 5.2 :

classe Personnage :

```
public class Personnage {
	// champs d'instances
	private int point_de_vie = 150;
	private int point_de_force = 200;
	public void marcher (){
		System.out.println("Je marche");
	}
	public void dormir (){
		System.out.println("Je dors");
	}
	public void manger (){
		System.out.println("Je mange");
	// Notre méthode affichage
	}
	public void affichage (){
		System.out.println("Point de vie = "+point_de_vie);
		System.out.println("Point de force = "+point_de_force);

	}
}
```

Classe Magicien

```
//Pour "dire" à Java que Magicien hérite de Personnage on utilise "extends"
public class Magicien extends Personnage {
	//champ d'instance
	private int point_de_magie = 200;
	public void faire_de_la_magie () {
		System.out.println ("Je fais de la magie");
	}
	public void affichage_magicien(){
		System.out.println ("Points de  magie = "+point_de_magie);
	}
}
```

Classe Principal

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// créons une instance de Magicien
		Magicien gandalf = new Magicien ();
		// ulisation de la méthode affichage_magicien (point_de_magie)
		gandalf.affichage_magicien();
		// ulisation de la méthode affichage (point_de_force et
point_de_vie)
		gandalf.affichage();
		}
}
```

Résultat :

```
Points de  magie = 200
Point de vie = 150
Point de force = 200
```

Tout fonctionne, pour afficher l'état de Gandalf, il suffit d'appeler la méthode *affichage_magicien* puis la méthode *affichage* (héritée de la classe *Personnage*). Si vous créez un simple personnage, il suffira d'appeler la méthode *affichage* (et uniquement la méthode *affichage*).

Bon, si vous réfléchissez un peu, tout cela n'est pas très satisfaisant : appeler 2 méthodes pour un magicien et une seule méthode pour un simple personnage, nous devons pouvoir faire mieux !!

Nous allons redéfinir la méthode *affichage*. Pour l'instant, il existe une méthode *affichage* qui appartient à la classe *Personnage*. Nous allons écrire une deuxième méthode *affichage* qui appartiendra à la méthode *Magicien* (à la place de la méthode *affichage_magicien*). Les 2 méthodes *affichage* auront exactement la même signature (nombre de paramètres et type de paramètres), c'est-à-dire pour nos 2 méthodes *affichage*, aucun paramètre. Attention, nous n'avons pas affaire ici à une surcharge de méthode (comme vu au chapitre II) mais à une redéfinition de la méthode *affichage* (dans la classe *Magicien* notre nouvelle méthode *affichage* remplace la méthode *affichage* héritée de la classe *Personnage*).

Pour ne pas vous tromper :

2 méthodes ayant le même nom mais pas les mêmes paramètres => surcharge
2 méthodes ayant le même nom et les mêmes paramètres => redéfinition

Réécrivons notre programme :

Ex 5.3

Classe Magicien

```
//Pour "dire" à Java que Magicien hérite de Personnage on utilise "extends"
public class Magicien extends Personnage {
	//champ d'instance
	private int point_de_magie = 200;
	public void faire_de_la_magie () {
		System.out.println ("Je fais de la magie");
	}
	// nouvelle méthode affichage
	public void affichage(){
		System.out.println("Point de vie = "+point_de_vie);
		System.out.println("Point de force = "+point_de_force);
		System.out.println ("Points de  magie = "+point_de_magie);
```

Et cela ne fonctionne pas !!

Nous avons le droit à une belle erreur :

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
	The field Personnage.point_de_vie is not visible
	The field Personnage.point_de_force is not visible
```

Nous avons oublié que les champs *point_de_vie* et *point_de_force* étaient des champs privés de la classe *Personnage*, ils sont donc inaccessibles depuis la classe *Magicien* !

Comment résoudre ce problème ?

Rendre les champs *public*, mais nous avons déjà vu que cela n'était pas très propre, ce n'est donc pas la solution.

Ce qu'il faudrait, c'est que la méthode *affichage* de la classe *Magicien* commence par appeler la méthode *affichage* de la classe *Personnage*. Et c'est ce que nous allons faire grâce au mot clé *super*.

Ex 5.4

Classe Magicien

```
//Pour "dire" à Java que Magicien hérite de Personnage on utilise "extends"
public class Magicien extends Personnage {
	//champ d'instance
	private int point_de_magie = 200;
	public void faire_de_la_magie () {
		System.out.println ("Je fais de la magie");
	}
	// nouvelle méthode affichage
	public void affichage(){
		// nous appelons la méthode affichage de la classe parente
(Personnage)
		super.affichage();
		// il ne nous reste plus qu'à afficher le champ point_de_magie
		System.out.println ("Points de  magie = "+point_de_magie);
	}
}
```

Classe Principal

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// créons une instance de Magicien
		Magicien gandalf = new Magicien ();
		gandalf.affichage();
		}
}
```

Classe Personnage inchangée

Cette fois tout fonctionne, nous avons bien utilisé la ligne *super.affichage ( )* pour appeler la méthode *affichage* de la classe *Personnage* (classe parente).

Notez qu'il est tout à fait possible de « mélanger » redéfinition et surcharge.

Pour vous entrainer, vous pouvez écrire une classe *Chevalier* qui héritera de la classe *Personnage*.
Un chevalier possède des points d'armure (nouveau champ d'instance : *point_armure*), mais aussi bien sûr, des points de vie et des points de force (comme tous nos personnages). De plus, la classe *Chevalier* devra aussi posséder sa propre méthode *affichage* (qui donnera les points de vie, les points de force et les points d'armure).

Constructeur et héritage

Si la classe B hérite de la classe A, B peut très bien utiliser le constructeur de la classe A. Pour appeler le constructeur de la classe parente, il faudra utiliser le mot clé *super*
=> *super (arg1, arg2....)*.
Ce mot clé *super* devra constituer la première ligne du constructeur de la classe dérivée (classe B).

Ex 5.5

Classe Personnage

```
public class Personnage {
	// champs d'instances
	private int point_de_vie;
	private int point_de_force;
	public Personnage (){
		System.out.println("Vous venez de créer un nouveau personnage");
	}
}
```

Classe Magicien

```
//Pour "dire" à Java que Magicien hérite de Personnage on utilise "extends"
public class Magicien extends Personnage {
	//champ d'instance
	private int point_de_magie;
	public Magicien(){
	// Appele du constructeur de la classe Magicien
	super();
	System.out.println("C'est un magicien");
	}
}
```

Classe Principal

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// créons une instance de Magicien
		Magicien gandalf = new Magicien ();
		}
}
```

Résultat :

```
Vous venez de créer un nouveau personnage
C'est un magicien
```

C'est un exemple très simple, je pense qu'il n'y a pas grand chose à dire.
Compliquons un peu les choses en passant des paramètres à nos constructeurs.

Ex 5.6

Classe Personnage

```
public class Personnage {
	// champs d'instances
	private int point_de_vie;
	private int point_de_force;
	public Personnage (int vie, int force){
		point_de_vie = vie;
		point_de_force= force;
		System.out.println("Vous venez de créer un nouveau personnage");
	}
	public void affichage (){
		System.out.println("point de vie = "+point_de_vie);
		System.out.println("point de force = "+point_de_force);

	}
}
```

Classe Magicien

```
public class Magicien extends Personnage {
	//champ d'instance
	private int point_de_magie;
	public Magicien(int vie, int force, int magie){
	super(vie, force);
	point_de_magie = magie;
	System.out.println("C'est un magicien");
	}
	public void affichage (){
		super.affichage();
		System.out.println("point de magie = "+point_de_magie);
	}
}
```

Classe Principal

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// créons une instance de Magicien
		Magicien gandalf = new Magicien (150,200,300);
		gandalf.affichage();
		}
}
```

Résultat :

```
Vous venez de créer un nouveau personnage
C'est un magicien
point de vie = 150
point de force = 200
point de magie = 300
```

Il est important de remarquer que le constructeur de la classe *Magicien* demande 3 arguments (*vie*, *force* et *magie*) alors que le constructeur de la classe *Personnage* n'en demande que deux (*vie* et *force*).
Les deux premiers arguments du constructeur de la classe *Magicien* (*vie* et *force*) sont « envoyés » au constructeur de la classe *Personnage* (1ère ligne : *super (vie, force)*). Le constructeur de la classe *Personnage* utilise ces 2 arguments pour initialiser les champs d'instance *point_de_vie* et *point_de_force*. Le 3ème argument du constructeur de la classe *Magicien* (*magie*) est utilisé directement dans le constructeur de la classe *Magicien* (`point_de_magie = magie;`)

Voilà, tout cela commence un peu à se compliquer, n'hésitez donc pas à travailler cet exemple et à en faire d'autres. Vous pourriez poursuivre l'écriture de la classe *Chevalier* en ajoutant un constructeur avec des arguments.

Notions de polymorphisme

Le polymorphisme est une notion plus difficile, nous allons juste voir un exemple.

Pour essayer de faire simple, on peut caractériser le polymorphisme en disant qu'il permet de manipuler des objets sans en connaître le type.

Reprenons les classes *Personnage* et *Magicien* définis dans l'exemple 5.6, modifions uniquement la classe *Principal*.

Ex 5.7

Classe Principal

```
public class Principal {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// Déclaration d'un objet de type Personnage (réservation d'une
		place en mémoire)
		Personnage gandalf;
		// Création d'une instance de type Personnage
		gandalf = new Personnage (150,200);
		// Appel de la méthode affichage de la classe Personnage
		gandalf.affichage();
		// Voila le polymorphisme
		System.out.println ("Faisons maintenant du polymorphisme !");
		gandalf = new Magicien (150,200,250);
		 // Appel de la méthode affichage de la classe Magicien
		gandalf.affichage();
		}
}
```

Résultat :

```
Vous venez de créer un nouveau personnage
point de vie = 150
point de force = 200
Faisons maintenant du polymorphisme !
Vous venez de créer un nouveau personnage
C'est un magicien
point de vie = 150
point de force = 200
point de magie = 250
```

Notre méthode *main* commence par la déclaration d'un objet de type *Personnage* (Java va réserver de la place en mémoire pour « accueillir » ce type d'objet (les champs d'instance *point_de_vie* et *point_de force*, le constructeur de la classe *Personnage* et la méthode *affichage* (version classe *Personnage*)).
Pourquoi avoir réservé de la place pour ce type d'objet ?
Tout simplement parce que le programme « devrait fabriquer » une instance de type *Personnage*, ce qu'il fait d'ailleurs à la ligne suivante :

```
gandalf = new Personnage (150,200);
```

Imaginons maintenant qu'au début de notre « jeu », Gandalf soit un simple personnage (voilà pourquoi nous créons une instance de type *Personnage*). Mais à force de travail, Gandalf a appris la magie, il est donc devenu magicien. Comment gérer cela au niveau de la programmation ?
Sans le polymorphisme nous devrions déclarer un autre objet de type *Magicien*, pour pouvoir créer une nouvelle instance de Gandalf.
Ici c'est inutile, Java s'adapte en « rangeant » une instance de type *Magicien* dans une case mémoire à l'origine réservée à un objet de type *Personnage.*
En gros le polymorphisme, c'est cela : déclarer un objet sans trop savoir ce qu'il va devenir !

Attention tout de même, tout cela est possible car nous avons une relation d'héritage entre la classe *Personnage* et la classe *Magicien*, le polymorphisme est donc intimement lié à l'héritage.

Voilà, je pense que vous en savez assez sur le sujet pour pouvoir reconnaître du polymorphisme quand vous en rencontrerez.

Chapitre 6

Les bases de la programmation graphique
1er partie : Fenêtres et boutons

Première fenêtre

Comme vous devez vous en douter une fenêtre est un objet. Nous allons utiliser la classe *JFrame* pour créer une instance *fen* (qui sera notre première fenêtre).

JFrame fen = new JFrame () ;

Par défaut notre fenêtre a une taille nulle. Nous allons utiliser la méthode *setsize (taille_x, taille_y)* fournie par la classe *JFrame* pour donner une taille à notre fenêtre.

fen.setsize (400,200) ;

Nous venons de créer une fenêtre de 400 pixels de large et de 200 pixels de haut.

Nous pouvons aussi donner un titre à notre fenêtre grâce à la méthode *setTitle ("Titre ")* (appartenant toujours à la classe *JFrame*).

fen.setTitle ("Première fenêtre");

Enfin, toujours par défaut, notre fenêtre est invisible, pour la faire apparaître à l'écran il faut utiliser la méthode *setVisible (true)*.

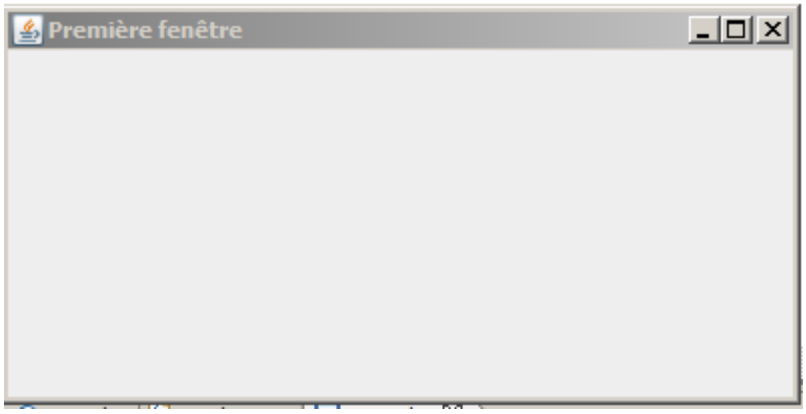
fen.setVisible (true) ;

Voilà, tout ceci a été regroupé dans l'exemple 6.1

Ex 6.1

```
// Nous avons besoin de la classe JFrame qui se trouve dans swing
import javax.swing.JFrame;
public class Fenetre {
      public static void main(String[] args) {
            // Création d'une instance de la classe JFrame (notre fenêtre)
            JFrame fen = new JFrame ();
            // Nous devons donner une taille à notre fenêtre (par défaut (0,0))
            fen.setSize (400,200);
            // Donnons un titre à notre fenêtre (non obligatoire)
            fen.setTitle("Première fenêtre");
            // Nous devons rendre notre fenêtre visible
            fen.setVisible(true);
      }
}
```

Résultat



Vous pouvez grâce à votre souris modifier la taille de la fenêtre, fermer la fenêtre.....
Tout cela est géré en interne par la classe *JFrame*, vous n'avez pas à vous en préoccuper (encore une fois c'est beau la POO !).

En fait, « fabriquer » une fenêtre dans la méthode *main*, cela ne se fait pas.....

Pourquoi ?

La méthode *main* doit être réservée au cœur du programme, imaginez un programme avec plusieurs fenêtres, votre méthode *main* va vite devenir illisible.

Nous allons créer une classe *Fenetre (*héritant de la classe *JFrame)* qui sera chargée de créer notre fenêtre. La méthode *main* sera uniquement utilisée pour créer une instance de notre classe *Fenetre* : *fen*.

Ex 6.2

Classe Fenetre

```
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
	public Fenetre (){
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
	}
}
```

Classe Principal

```
public class Principal {
	public static void main(String[] args) {
		Fenetre fen;
		fen = new Fenetre();
		fen.setVisible(true);

	}

}
```

Le mot clé *this*

Vous avez sans doute remarqué ce mot clé mystérieux : *this*
Dans cet exemple, *this* « remplace » notre instance *fen*, je m'explique :
Java commence par exécuter la méthode *main*, il crée donc une instance de la classe *Fenetre* : *fen*. Immédiatement après la création de *fen*, le constructeur de la classe *Fenetre* est appelé. Pour essayer de comprendre la fonction du mot clé *this*, remplacer *this* par *fen*.
On a donc *fen.setTitle....*, *fen.setSize*..... Les méthodes *setTitle*, *setSize* et *setVisible* sont appliquées à l'instance *fen*.
Vous allez me dire : « Pourquoi ne pas mettre *fen* à la place de *this* ? »
Au moment de la création de votre classe *Fenetre*, vous ne savez pas comment s'appellera l'instance créée dans la méthode *main* ! Si nous avions appelé notre instance *toto*, ***this***.*setSize* aurait été l'équivalent de *toto.setSize*.
Nous pouvons donc dire que l'opérateur *this* sert à référencer l'objet en cours.

Il faut savoir que dans ce cas particulier, *this* peut-être sous entendu. Nous pouvons donc avoir :

Classe Fenetre

```
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            setTitle ("Titre de la fenêtre");
            setSize (400,200);
      }
}
```

Pour Java `setTitle ("Titre de la fenêtre");` est équivalent à `this.setTitle ("Titre de la fenêtre");`
Ne soyez donc pas surpris si vous rencontrez des méthodes « qui se baladent toutes seules » comme dans le dernier exemple, en fait elles ne sont pas « seules », il y a le mot clé *this* devant. Après, vous faites comme vous voulez, à vous de choisir (avec le *this* ou sans le *this*).

Premier bouton

Une fenêtre vide ce n'est pas mal, mais il va falloir songer à la remplir!! Nous allons maintenant ajouter notre premier composant : un bouton.

Tous les composants d'une fenêtre doivent se trouver dans un conteneur (sorte de support qui va accueillir les éléments : bouton,....), il faut donc créer une instance de la classe *Container*.
La méthode *getContentPane* (qui appartient à la classe *JFrame*) renvoie (avec l'instruction *return*, mais bon, grâce à la magie de la POO, ce n'est pas vraiment notre problème) un objet de type *Container*. Nous aurons donc :
Container cont = this.getContentPane ( )
On applique la méthode *getContentPane* à l'instance courante (*this*) c'est à dire ici à l'instance *fen*, nous créons donc un objet *Container* dans notre fenêtre.

Ajoutons maintenant notre bouton :
Un bouton est une instance de la classe *JButton*, nous allons donc créer notre instance :
JButton notre_bouton
notre_bouton = new JButton (" Chouette, cela fonctionne")

Nous pouvons aussi écrire directement :
JButton notre_bouton = new JButton (" Chouette, cela fonctionne")

Nous devons ensuite « ajouter » notre bouton dans notre conteneur grâce à la méthode *add* de la classe *Container.*
cont.add (notre_bouton)

Je pense que vous avez compris que cette méthode *add* prend comme paramètre l'objet à ajouter.

Si nous regroupons tout cela dans notre classe *Fenetre*, cela donne :

Ex 6.3

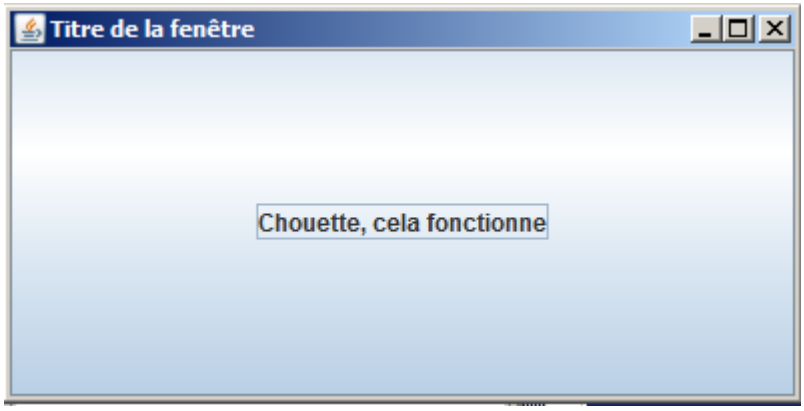
Classe Principal : inchangée par rapport à l'exemple 6.2

Classe Fenetre

```
//Nous devons importer la classe Container (package awt)
import java.awt.Container;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            this.setTitle ("Titre de la fenêtre");
            this.setSize (400,200);
            // Création de notre conteneur
            Container cont = this.getContentPane ();
            // Création de notre bouton
            JButton notre_bouton = new JButton ("Chouette, cela fonctionne");
            cont.add(notre_bouton);
      }
}
```

Cela fonctionne, même si cela n'en a pas l'air !

En fait, notre bouton occupe toute la fenêtre.



Il faut savoir que la disposition des composants dans une fenêtre est gérée par un « gestionnaire de mise en forme » (en anglais « layout manager »). Il existe plusieurs gestionnaires possibles (chaque gestionnaire est une classe différente).
Par défaut, Java utilise le gestionnaire *BorderLayout* (notez le B majuscule, c'est bien une classe).

Sans intervention du programmeur, avec ce gestionnaire, le composant (notre bouton) occupe toute la place disponible, c'est-à-dire toute la fenêtre.

Il existe un gestionnaire de mise en forme (le gestionnaire *FlowLayout*) qui dispose les composants les uns à la suite des autres, d'abord sur une même ligne, puis ligne par ligne.

Pour choisir ce layout manager, il suffit d'appliquer la méthode *setLayout* au conteneur (ici notre instance *cont*). Cette méthode a besoin d'une instance de la classe *FlowLayout* :

cont.setLayout (new FlowLayout ( ))

Nous n'aurons pas à réutiliser l'objet de type *FlowLayout*, voilà pourquoi nous passons en paramètre directement *new FlowLayout ( ),* sans le déclarer dans une variable.

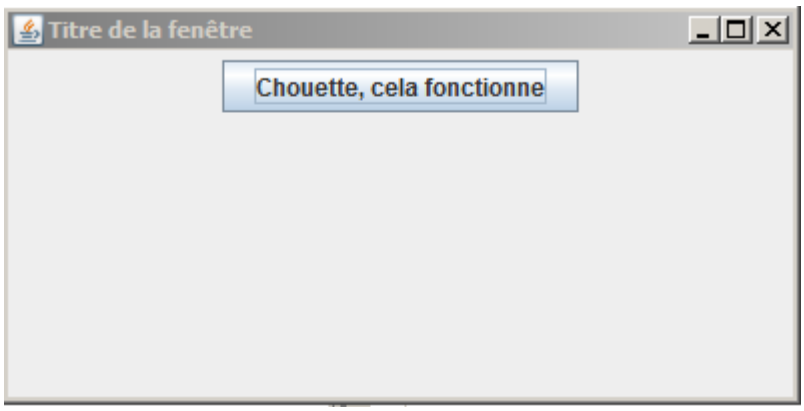
Ex 6.4

La classe *Principal* reste inchangée

Classe Fenetre

```
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            this.setTitle ("Titre de la fenêtre");
            this.setSize (400,200);
            // Création de notre container
            Container cont = this.getContentPane ();
            // Choix du layout manager
            cont.setLayout (new FlowLayout());
            // Création de notre bouton
            JButton notre_bouton = new JButton ("Chouette, cela fonctionne");
            cont.add(notre_bouton);
      }
}
```

Résultat :



Voilà, notre bouton ne prend plus toute la place.

En général, les programmeurs sont un peu fainéants, ils essaient d'écrire le moins de code possible. Ce code peut se simplifier :

Ex 6.5

Classe Fenetre

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            this.setTitle ("Titre de la fenêtre");
            this.setSize (400,200);
            // Choix du layout manager
            this.getContentPane ().setLayout (new FlowLayout());
            // Création de notre bouton
            JButton notre_bouton = new JButton ("Chouette, cela fonctionne");
            // Ajout du bouton
            this.getContentPane ().add(notre_bouton);
      }
}
```

Qu'avons-nous modifié ?

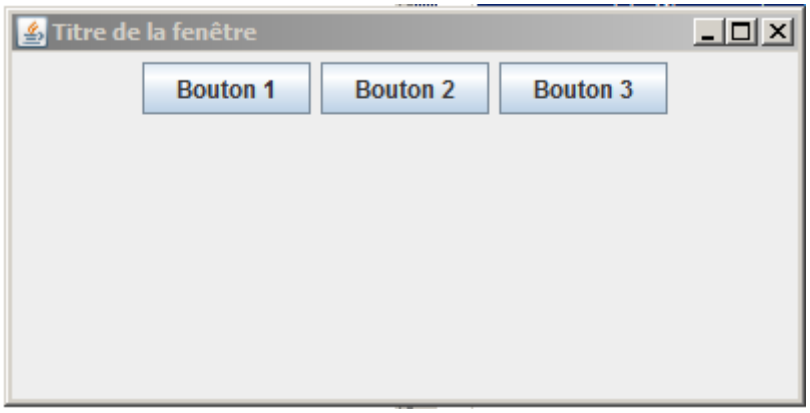
Vous avez sans doute remarqué la disparition de notre instance *cont* de la classe *Container*. Tout simplement, cette instance *cont* est inutile.
Rappelez-vous, la méthode *getContentPane* « fabrique » un objet de type *Container*. Ici, l'idée c'est de créer cet objet et de l'utiliser dans la foulée. La méthode *getContentPane ()* crée l'objet qui est immédiatement utilisé par la méthode *setLayout* (sur la même ligne). C'est exactement la même chose avec la méthode *add* à la dernière ligne.
Vous n'êtes pas obligé de simplifier les choses, rien ne vous empêche d'utiliser une instance de type *Container* (comme l'instance *cont* dans l'exemple 6.4), parfois vous serez même obligés de l'utiliser.

Nous allons maintenant créer un deuxième et un troisième bouton.

Ex 6.6

```
//Nous devons importer la classe Container (package awt)
import java.awt.*;
import javax.swing.JFrame;
import javax.swing.JButton;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            this.setTitle ("Titre de la fenêtre");
            this.setSize (400,200);
            // Création de notre container
            Container cont = getContentPane ();
            // Choix du layout manager
            cont.setLayout (new FlowLayout());
            // Création de notre bouton
            JButton bouton1 = new JButton ("Bouton 1");
            cont.add(bouton1);
            JButton bouton2 = new JButton ("Bouton 2");
            cont.add(bouton2);
            JButton bouton3 = new JButton ("Bouton 3");
            cont.add(bouton3);
      }
}
```

Résultat :



Rien à ajouter, cela fonctionne.

Il existe d'autres gestionnaires de mise en forme, nous allons en voir un second : Le *BorderLayout*

Un exemple vaut mieux qu'un long discours :

Ex 6.7

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
      public Fenetre (){
            this.setTitle ("Titre de la fenêtre");
            this.setSize (400,200);
            // Création de notre container
            Container cont = getContentPane ();
            // Création de notre bouton
            JButton bouton1 = new JButton ("Bouton 1");
            cont.add(bouton1,BorderLayout.NORTH);
            JButton bouton2 = new JButton ("Bouton 2");
            cont.add(bouton2,BorderLayout.SOUTH);
            JButton bouton3 = new JButton ("Bouton 3");
            cont.add(bouton3,BorderLayout.WEST);
            JButton bouton4 = new JButton ("Bouton 4");
            cont.add(bouton4,BorderLayout.EAST);
            JButton bouton5 = new JButton ("Bouton 5");
            cont.add(bouton5,BorderLayout.CENTER);
      }
}
```

Résultat :

Je pense que ce résultat est parlant, le choix du gestionnaire de mise en forme se fait au moment de l'ajout du bouton dans le container, notre méthode *add* a maintenant deux paramètres : le nom de l'instance qui correspond à notre bouton et *BorderLayout.NORTH* (pour le premier exemple)

Il faut ici faire une petite parenthèse : à quoi correspond ce *BorderLayout.NORTH* ?

C'est la première fois que nous rencontrons un mot écrit exclusivement en majuscule (*NORTH*). Par convention, en Java, les constantes sont écrites en majuscule.

Une constante, est une sorte de variable dont la valeur n'est pas modifiable (oui, je sais, c'est paradoxal une variable qui ne change pas !). Une constante est définie comme une variable, on ajoute simplement le mot final pour bien faire comprendre à Java que la valeur contenue dans cette « variable » ne devra en aucun cas être modifiée par la suite.

final int TOTO = 15
Dans la suite du programme *TOTO* sera égale à 15, cette valeur est définitive.

Une constante peut-être de type *int*, mais aussi de type *String*, de type *double*,..... comme les variables.

Il existe, tout comme pour les variables, des constantes de classe, déclarées avec le mot clé *static*.

Pour utiliser une constante de classe on devra avoir une expression de la forme :
NomDeMaClasse.CONSTANTE

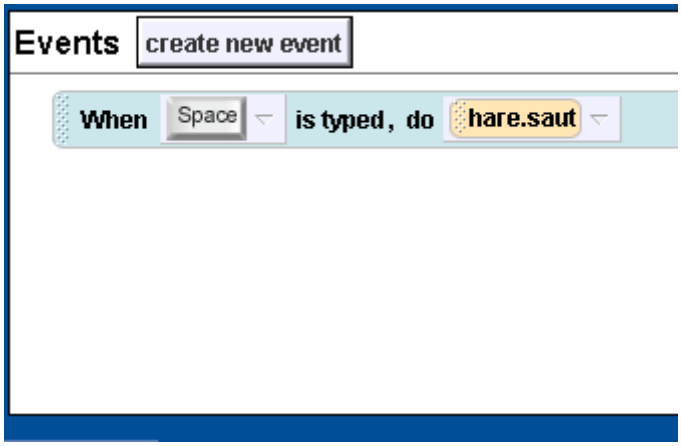
Si vous réfléchissez un peu, c'est exactement ce que l'on a dans notre exemple avec :
BorderLayout.NORTH

BorderLayout est une classe, *NORTH* est la constante *static*.

Chapitre 7
Les bases de la programmation graphique
2eme partie : Gestion des évènements, les listeners

Pour l'instant nos fenêtres ne font strictement rien, en cas de clic (sur un bouton ou dans la fenêtre) , rien ne se passe !

Nous allons donc devoir écrire du code pour gérer toutes les actions possibles (clic de souris, appui sur une touche,.....). Nous avons déjà eu à faire ce genre de choses dans Alice avec le gestionnaire d'événements.



Dans ce programme écrit avec Alice, le lièvre (hare) ne fait rien. En revanche, si l'utilisateur appuie sur la touche « Espace », on aura alors l'appel de méthode saut.
Dans Alice, le gestionnaire d'événements « surveille » la touche « Espace » et réagit en cas d'appuis sur cette dernière.

Nous allons avoir exactement la même chose en Java avec les listeners (en français écouteurs !)

Il existe différents types de listener, nous allons dans un premier temps étudier le *MouseListener* (qui va permettre de « surveiller » la souris).

Mais qu'est ce qu'un listener ?

Mouselistener est une interface comme tous les listeners, mais qu'est-ce qu'une interface ?

Dans certains langages (comme le C++ par exemple), il est possible de faire de l'héritage multiple (une classe peut directement hériter de plusieurs classes), cela peut-être pratique mais c'est terriblement compliqué à utiliser correctement. Java n'autorise pas l'héritage multiple, ce concept est remplacé (avantageusement ?) par les interfaces.
En deux mots et sans trop entrer dans les détails, une interface est une sorte de classe composée de méthodes vides !
Passons à un exemple :

Pour déclarer une interface il suffit de remplacer le mot *class* par le mot *interface* :

```
public interface MonInterface  {
        // déclaration des méthodes qui composent notre interface MonInterface
        void maMethode1 ( );
        void maMethode2 (arg);
.
.
.
}
```

Voilà nous avons uniquement donné le nom des méthodes et éventuellement leurs arguments, on dit que nous avons donné les en-têtes des méthodes (valeur de retour + nom de la méthode + arguments).
Notre interface *Moninterface* peut-être implémentée dans n'importe quelle classe à l'aide du mot clé *implements*.

```
public class MaClasse implements MonInterface {
.
.
.
}
```

Pour ne pas avoir d'erreur il faudra obligatoirement que notre classe *Maclasse* possède toutes les méthodes de l'interface *MonInterface* (*maMethode1* et *maMethode2*) .

```
public class MaClasse implements MonInterface {
        public void maMethode1 () {
                ……..
        }
        public void maMethode2 (arg) {
                …..
        }
}
```

Rien ne vous empêche de laisser une méthode vide

```
public class MaClasse implements MonInterface {
        public void maMethode1 () { }
        public void maMethode2 (arg) {
                …..
        }
}
```

Il y aurait beaucoup d'autres choses à dire sur les interfaces, mais je pense que vous en savez assez pour continuer notre étude des listeners.

Comme déjà dit plus haut *Mouselistener* est une interface (nous verrons un peu plus loin les méthodes qui composent cette interface), elle doit donc être implémentée dans une classe.

Nous allons créer une nouvelle classe *Ecouteur_souris*, les instances de cette classe seront chargées de « surveiller » la souris et de réagir en cas d'action de l'utilisateur sur cette dernière. Notre classe *Ecouteur_souris* doit implémenter l'interface *Mouselistener.*

Il est très important de bien comprendre que c'est un objet (une instance) qui sera chargé de la surveillance de notre souris.

Ex 7.1

Classe Ecouteur_souris

```
// attention de bien importer MouseEvent et MouseListener
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class Ecouteur_souris implements MouseListener {
	// Voilà toutes les méthodes (vide) de l'interface MouseListener
	public void mouseClicked(MouseEvent even){}
	public void mouseEntered(MouseEvent even){}
	public void mouseExited(MouseEvent even) {}
	public void mousePressed(MouseEvent even){}
	public void mouseReleased(MouseEvent even){}
}
```

Vous pouvez remarquer que les méthodes implémentées demandent une instance (*even*) de type *MouseEvent*.

Classe Fenetre

```
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
	public Fenetre (){
		//Créons une instance de la classe Ecouteur_souris
		Ecouteur_souris ecou = new Ecouteur_souris ();
		// La méthode addMouseListener permet de mettre la fenêtre sur
		"écoute"
		this.addMouseListener (ecou);
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
	}
}
```

Nous avons ajouté 2 lignes (et enlevé les lignes qui concernent les boutons, provisoirement) :

`Ecouteur_souris ecou = new Ecouteur_souris ();`
Crée une instance de notre nouvelle classe.

`this.addMouseListener (ecou);`
Cette ligne permet de mettre « sur écoute » l'instance courante (*this*), c'est à dire ici notre fenêtre (*fen*).

La classe *Principal* n'est pas modifiée

Nous voulons que le programme réagisse au clic de souris, nous allons donc modifier notre classe *Ecouteur_souris* et plus précisément la méthode *mouseClicked*

Ex 7.2

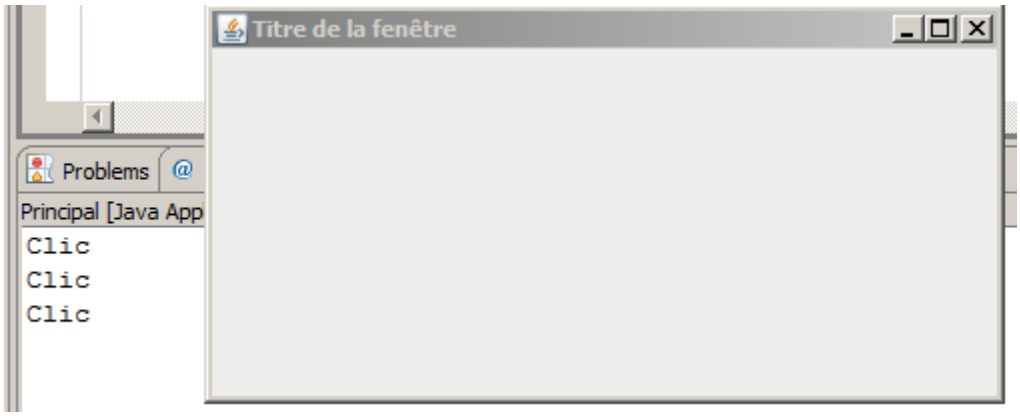
Classe Ecouteur_souris

```
// attention de bien importer MouseEvent et MouseListener
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class Ecouteur_souris implements MouseListener {
	// Nous avons complété la méthode mouseClicked
	public void mouseClicked(MouseEvent even){
		System.out.println("Clic");
	}
	// Les autres méthodes restent inchangées
	public void mouseEntered(MouseEvent even){}
	public void mouseExited(MouseEvent even) {}
	public void mousePressed(MouseEvent even){}
	public void mouseReleased(MouseEvent even){}
}
```

Les classes *Principal* et *Fenetre* restent inchangées

Résultat



A chaque clic dans la fenêtre, le mot « Clic » apparaît dans la console.

Analysons ce qui se passe.

Dès le démarrage de notre programme, un listener est créé, il est chargé de surveiller la souris. A chaque clic, le listener appelle la méthode *mouseClicked*, cette méthode affiche le mot « Clic » dans la console d'Eclipse.

Voilà, je pense que vous devez avoir compris le fonctionnement de *MouseListener*. Je vous propose tout de même un autre exemple qui vous permettra de comprendre l'utilité des autres méthodes de l'interface *MouseListener*.

Ex 7.3

Classe Ecouteur_Souris

```
// attention de bien importer MouseEvent et MouseListener
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class Ecouteur_souris implements MouseListener {
	// Nous avons complété toutes les méthodes
	public void mouseClicked(MouseEvent even){
		System.out.println("Clic");
	}

	public void mouseEntered(MouseEvent even){
		System.out.println("Entrée dans la fenêtre");
	}
	public void mouseExited(MouseEvent even) {
		System.out.println("Sortie de la fenêtre");
	}
	public void mousePressed(MouseEvent even){
		System.out.println("Presse");
	}
	public void mouseReleased(MouseEvent even){
		System.out.println("Relâche");

	}
}
```

Les classes *Principal* et *Fenetre* restent inchangées

Nous ne sommes pas obligés de créer un objet écouteur. L'objet écouteur peut être n'importe quel objet à condition que sa classe implémente l'interface voulue (ici *MouseListener*). Dans l'exemple suivant nous allons utiliser la fenêtre comme objet écouteur de …... la fenêtre (elle va faire de l'autosurveillance !)

Ex 7.4

Classe Fenetre

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame et implémente MouseListener
public class Fenetre extends JFrame implements MouseListener {
	public Fenetre (){
		// La méthode addMouseListener permet de mettre la fenêtre sur
		"écoute"
		// elle utilise l'instance fen (d'où le this) comme objet écouteur
		this.addMouseListener (this);
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
	}
	public void mouseClicked(MouseEvent even){
		System.out.println("Clic");
	}

	public void mouseEntered(MouseEvent even){
		System.out.println("Entrée dans la fenêtre");
	}
	public void mouseExited(MouseEvent even) {
		System.out.println("Sortie de la fenêtre");
	}
	public void mousePressed(MouseEvent even){
		System.out.println("Presse");
	}
	public void mouseReleased(MouseEvent even){
		System.out.println("Relâche");

	}
}
```

Classe *Principal* inchagé, plus de classe *Ecouteur_souris*

Quelques remarques :

- Nous avons bien une « auto écoute » : `this.addMouseListener (this);`
  Le premier *this* correspond à l'objet écouté, le deuxième à l'objet écouteur (je vous rappelle que rien ne vous empêche de simplifier cette ligne en supprimant le premier *this* (qui est alors sous entendu) : `addMouseListener (this))`
- Toutes les méthodes de l'interface *MouseListener* se trouvent maintenant dans la classe *Fenetre*.
- Il ne faut surtout pas oublier d'implémenter *MouseListener* à la classe *Fenetre* :
  public class Fenetre extends JFrame implements MouseListener

Si votre interface comporte de nombreuses méthodes, cela peut rapidement devenir lourd à gérer. Imaginez une interface avec 15 méthodes : cela fait 15 méthodes à réécrire dans la classe qui implémente cette interface, si en plus vous n'utilisez qu'une seule de ces 15 méthodes !?

Il existe donc une autre possibilité : l'utilisation d'un adaptateur.

Il existe une classe *MouseAdapter* qui implémente toutes les méthodes de l'interface *MouseListener*. Comme son nom l'indique *MouseAdapter* est un adaptateur.
Voici à quoi ressemble notre classe *MouseAdapter* (sans les *import*)

class MouseAdapter implements MouseListener {

```
	public void mouseEntered(MouseEvent even){}
	public void mouseExited(MouseEvent even) {}
	public void mousePressed(MouseEvent even){}
	public void mouseReleased(MouseEvent even){}
}
```

Réécrivons notre classe *Ecouteur_souris* :

Ex 7.5

Classe Fenetre

```
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame et implémente MouseListener
public class Fenetre extends JFrame {
	public Fenetre (){
		// on utilise l'instance ecou comme objet écouteur
		Ecouteur_souris ecou = new Ecouteur_souris ();
		this.addMouseListener (ecou);
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
	}
}
```

Classe Ecouteur_souris

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
public class Ecouteur_souris extends MouseAdapter {
	// Voilà toutes les méthodes (vide) de l'interface MouseListener
	public void mouseClicked(MouseEvent even){
		System.out.println("Clic");
	}
}
```

La classe *Principal* reste identique

Nous avons donc ici une redéfinition de la méthode *mouseClicked* de la classe *MouseAdapter*. L'utilisation d'un adaptateur (ici *MouseAdapter*) nous évite de réécrire toutes les méthodes de l'interface *MouseListenner*, le code est plus compact et plus lisible.

Le problème est que, visiblement, l'utilisation d'un adaptateur n'est pas possible dans le cas où la fenêtre est son propre écouteur (exemples 7.3 et 7.4). Pourquoi ?

En Java l'héritage multiple n'existe pas, notre classe *Fenetre* hérite déjà de la classe *JFrame* et ne peut donc pas hériter de la classe *MouseAdapter*.

Il existe pourtant une solution (comme toujours) : les classes anonymes.

Voici un exemple d'utilisation d'une classe anonyme

Ex 7.6

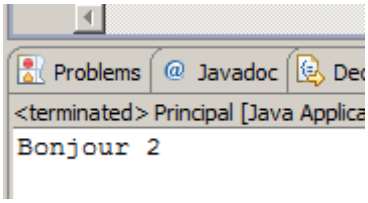
Classe Toto

```
public class Toto {
	public void affichage (){
		System.out.println ("Bonjour 1");
	}
}
```

Classe Principal

```
public class Principal {
	public static void main(String[] args) {
		Toto inst;
		inst = new Toto (){
			public void affichage () {
				System.out.println("Bonjour 2");
			}
		};
		inst.affichage();
	}
}
```

Résultat :



Nous avons notre classe Toto qui comporte une seule méthode : *affichage ( )* (cette méthode affiche « Bonjour 1 » dans la console).
La méthode *main* se trouve dans la classe *Principal* :
Nous commençons notre méthode *main* en créant une instance de la classe *Toto* (*inst*).
Remarquez l'accolade ouvrante qui suit le « new Toto ( ) {», elle est inhabituelle. C'est entre cette accolade ouvrante et l'accolade fermante qui est suivie d'un point virgule (attention de ne surtout pas oublier ce point virgule), que nous allons redéfinir une (ou les) méthode(s) de notre classe *Toto* (ici la méthode *affichage ( )*).
Le résultat nous montre que la méthode *affichage ( )* a bien été redéfini (affichage de « Bonjour 2 » et non pas de « Bonjour 1 ») sans pour autant que la classe *Principal* ait hérité de la classe Toto.

Nous allons redéfinir la méthode *mouseClicked* de la classe *MouseAdapter* sans que notre classe *Fenetre* hérite de la classe *MouseAdapter* (puisqu'elle hérite déjà de la classe *JFrame*).

Ex 7.7

Classe Fenetre

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame {
	public Fenetre (){
		this.addMouseListener (new MouseAdapter () {
			public void mouseClicked(MouseEvent even){
				System.out.println("Clic");
			}
		});
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
	}
}
```

Classe Principal (ex 7.5) inchangée.

Le paramètre de la méthode *addMouseListener* est un objet de type *MouseAdapter*. La méthode *mouseClicked* est rédéfinie. C'est donc l'objet de type *MouseAdapter* qui est chargé de la surveillance de la fenêtre (cela commence à être compliqué !)

Surveillance d'un bouton

Un bouton ne peut déclencher qu'un seul événement : appuie sur le bouton (à l'aide de la souris ou en appuyant sur la touche entrée).

L'interface à utiliser se nomme *ActionListener*, elle possède une seule méthode (puisque qu'il y a un seul événement possible) : *actionPerformed*
Pour associer l'écouteur à un bouton, la méthode *addActionListener* doit être utilisée.

Ex 7.8
Classe Fenetre

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame implements ActionListener {
	public Fenetre (){
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
		// Création de notre container
		Container cont = getContentPane ();
		// Choix du layout manager
		cont.setLayout (new FlowLayout());
		// Création de notre bouton
		JButton bouton = new JButton ("Appuyer Ici");
		cont.add(bouton);
		bouton.addActionListener(this);
	}
	public void actionPerformed (ActionEvent even){
		System.out.println("Vous avez appuyé sur le bouton");
	}
}
```

Classe Principal inchangée.

Je pense que cet exemple ne devrait pas vous poser de problème.

Comment surveiller 2 boutons ?

Nous allons utiliser la méthode *getSource* (méthode appartenant à la classe *ActionEvent*) qui permet de « reconnaitre » le bouton qui vient d'être actionné.

Ex 7.9

Classe Fenetre

```
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
// La classe Fenetre hérite de JFrame
public class Fenetre extends JFrame implements ActionListener {
	private JButton bouton1;
	private JButton bouton2;
	public Fenetre (){
		this.setTitle ("Titre de la fenêtre");
		this.setSize (400,200);
		// Création de notre container
		Container cont = getContentPane ();
		// Choix du layout manager
		cont.setLayout (new FlowLayout());
		// Création de 2 boutons avec les listeners
		bouton1 = new JButton ("Bouton1");
		cont.add(bouton1);
		bouton1.addActionListener(this);
		bouton2 = new JButton ("Bouton2");
		cont.add(bouton2);
		bouton2.addActionListener(this);
	}
	public void actionPerformed (ActionEvent even){
		if (even.getSource()== bouton1){
			System.out.println("Vous avez appuyé sur le bouton 1");
		}
		if (even.getSource()== bouton2){
			System.out.println("Vous avez appuyé sur le bouton 2");
		}
	}
}
```

La méthode *getSource* appliquée à l'instance *even* renvoie le nom de l'instance qui vient d'être actionnée :
Si vous appuyez sur le Bouton 1, la méthode *getSource* renvoie « bouton1 », le premier if renvoie donc *true*, on a donc l'affichage suivant dans la console : `"Vous avez appuyé sur le bouton 1"`

Si vous appuyez sur le Bouton 2, la méthode *getSource* renvoie « bouton2 », le premier if renvoie *false*, le second if renvoi *true*, on a donc l'affichage suivant dans la console : `"Vous avez appuyé sur le bouton 2"`

Il reste encore beaucoup de choses à voir en Java, mais vous en savez assez pour pouvoir aborder la programmation d'applications sous android (voir « De Java à Android »). Pour ceux qui voudraient aller plus loin en Java, je vous donne ici deux pistes :

livre :

Programmer en Java de Claude Delannoy (On trouve cet ouvrage pour moins de 20 euros en réédition dans la collection « Best Of » chez Eyrolles)

site internet :

le site du Zero : http://www.siteduzero.com/tutoriel-3-10601-programmation-en-java.html