



Initiation au langage JAVA

TABLE DES MATIERES

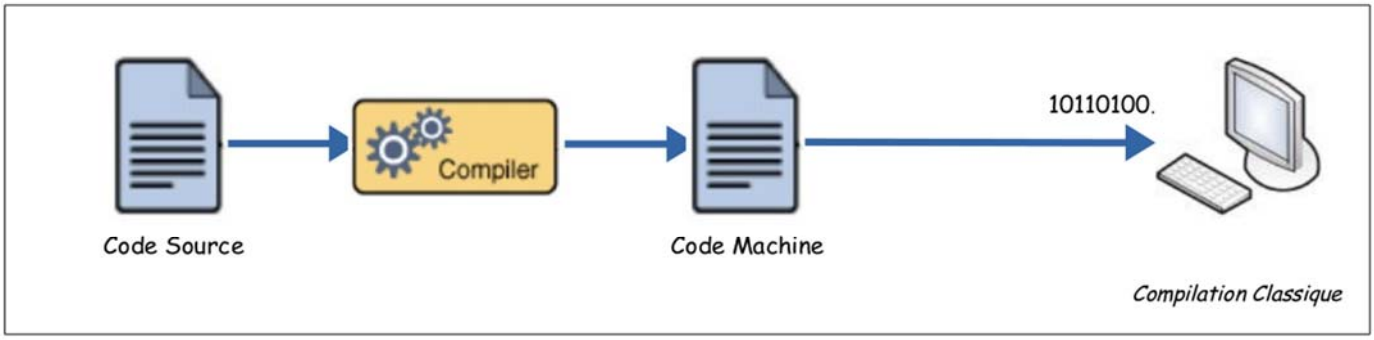
Notions de Base 5
1. Introduction 5
2. Caractéristique du langage Java 5
3. Bases de la programmation en JAVA 6
4. Environnement JAVA 6
5. Ligne de Commandes 7
6. Variables et Opérations 8
7. Fonction d'affichage et de saisie 11
8. Conversion de Type (Cast) 12
9. Structure de contrôle 13
TD n°1 : Comptage de bouteilles 14
10. Les Tableaux 15
Les Méthodes 17
1. Syntaxe d'une méthode 17
TD n°2 : Nombre mystère 18
Les classes et les objets en Java 19
1. Les classes et les objets en Java 19
2. La variable This 24
Exemple : Création d'un objet en JAVA 25
TD n°3 : Jeux sur les caractères 27
Initialisation d'un objet et constructeur 28
1. Initialisation d'un objet 28
Les Exceptions 29
1. Classes d'exception 29
2. Try, Catch et Finally 30
3. Throws, Throw 30
Les Fichiers 32
1. Les Exceptions 32
2. File 32
3. BufferedInputStream - BufferedOutputStream 33
4. RandomAccessFile 33
TD n°4 : Cryptographie 33

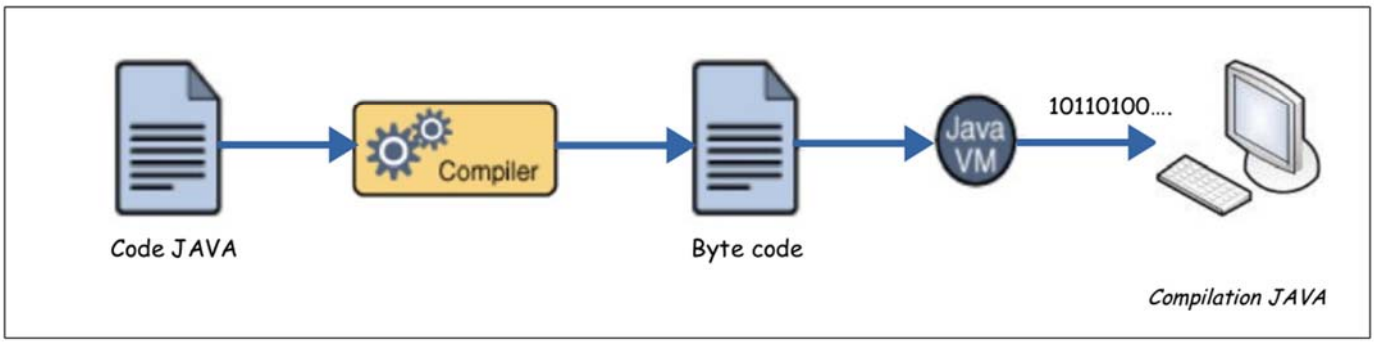
Notion sur la composition et l'agrégation en java 34
1. La composition 34
1.3. L'agrégation 36
L'héritage 38
1. Principe 38
2. Construction et initialisation des objets dérivés 40
3. Redéfinition de méthodes 42
TD N°5 L'héritage en JAVA 42
Initiation à la programmation graphique 46
TD N°6 : L'héritage pour la programmation graphique (Création d'une fenêtre et d'un panneau) 47
Ressources sur le dessin : la classe Graphics 50
1. Solution 1 : redéfinission de la méthode paintComponent 50
TD N°7 : Les composants graphiques 51
La programmation événementielle en JAVA : Notion d'interfaces et de classes abstraites 56
1. Les classes et les méthodes du type final 56
2. Les classes abstraites 56
3. Les interfaces 58
TD N°8 : Les interfaces 60
4. Les interfaces et la programmation événementielle 65
4.2. L'interface ActionListener 66
TD N°9 :Application: Interface ActionListener 67
TD N°10 : Synthèse étude d'un cas : L'AIS 69
Les threads ou processus léger 75
1. Rappel les processus lourds 75
2. Les processus légers 76
3. La nécessité d'un thread 78
4. Les principaux états d'un thread 81
5. Définition création et démarrage d'un thread 81
Les sockets et la programmation réseau 86
1. Introduction 86
2. La classe InetAddress 86
3. L'utilisation du protocole TCP 87
4. L'utilisation du protocole UDP (user datagram protocol) 96
Initiation aux bases de données relationnels 98
1. Définition 98
2. Le vocabulaire 98
3. Le langage SQL 99
4. JAVA et le langage SQL 99
Ressource sur le système AIS (Automatic Identification System) 105
1. Présentation 105
2. Le protocole AIS 105
3. Etude d'un message de type 1 106

Notions de Base

1. Introduction

Le langage Java est un des langages de programmation informatiques orientés objet le plus répandu dans le monde. Il a été créé par James Gosling et Patrick Naughton, employés de Sun Microsystems (racheté par Oracle), en 1995.





La machine virtuelle effectue la traduction et l'exécution du bytecode en code natif. On parle alors de langage semi-compilé ou semi-interprété.

2. Caractéristique du langage Java

- Simple (syntaxe proche du C)
- Indépendant de la plateforme (Window's, Linux, Sun-OS, Tini-OS...)
- Orienté Objet (des classes, des méthodes, de l'héritage...)
- Multithread (fonctionnalités natives pour gérer le multitâche)
- Économe (le pseudo code a une taille relativement petite car les bibliothèques de classes requises ne sont liées qu'à l'exécution)
- Sécurisé (protégé de l'extérieur et protège l'extérieur)

3. Bases de la programmation en JAVA

En ce qui concerne les types et instructions de base, le langage Java est proche du langage C. Toutefois, un certain nombre de différences notables existent : absence de pointeurs et de structures, gestion dynamique de la mémoire entièrement automatique…

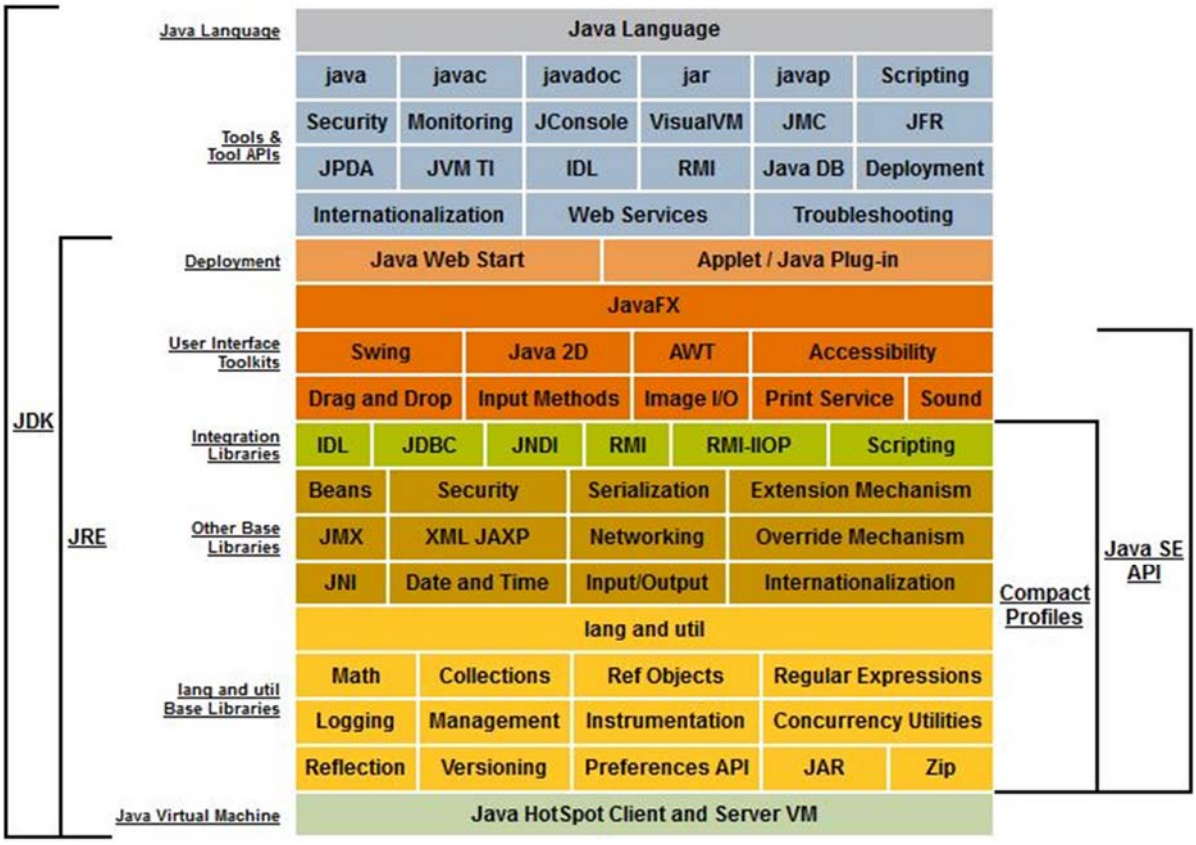
```
public class test {

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

    }

}
```

4. Environnement JAVA



Java SE Runtime Environment (JRE) : contient les librairies, JVM et d'autres composants pour lancer les applets et programmes écrit en JAVA sur des PC ou des serveurs. Il existe d'autres environnements JAVA en fonction de son rôle final : JEE pour les pages Web, JME pour les systèmes embarqués (smartphones)…

Java SE Development Kit (JDK) : inclue JRE avec en plus les outils pour les développeurs (compliers, debuggers).

Java Virtual Machines (JVM) : est une machine virtuelle qui exécute le bytecode JAVA. Celle-ci existe pour différentes plateformes.

Pour la suite du cours nous utiliserons l'IDE Eclipse qui possède tous les outils pour créer des applets et programmes JAVA (cf Tutoriel Eclipse).



5. Ligne de Commandes

- **Javac**

Cet outil est le compilateur, il transforme un ou plusieurs fichiers source JAVA en fichiers contenant le bytecode. Les fichiers source ont l'extension .java et peuvent s'ouvrir avec un simple éditeur de texte.
La syntaxe est la suivante :

javac [options] [fichiers] [@fichiers]

Option	Rôle
-classpath path	permet de préciser le chemin de recherche des classes nécessaires à la compilation -d répertoire
-d répertoire	les fichiers sont créés dans le répertoire indiqué. Par défaut, les fichiers sont créés dans le même répertoire que leurs sources.
-g	génère des informations de débogage
-nowarn	le compilateur n'émet aucun message d'avertissement
-O	le compilateur procède à quelques optimisations. La taille du fichier généré peut augmenter. Il ne faut pas utiliser cette option avec l'option -g
-verbose	le compilateur affiche des informations sur les fichiers sources traités et les classes chargées
-deprecation	donne des informations sur les méthodes dépréciées qui sont utilisées

- **Java**

Cet outil est l'interpréteur de bytecode, celui-ci lance le JRE, charge les classes nécessaires et exécute la méthode mais de la classe passée en paramètre.
La syntaxe est la suivante :

java [options] classe [argument ...]

Option	Rôle
-jar archive	Permet d'exécuter une application contenue dans un fichier .jar. Dans ce cas, le fichier manifest de l'archive doit préciser quelle est la classe qui contient la méthode main().
-Dpropriete=valeur	Permet de définir une propriété système sous la forme propriete=valeur. propriété représente le nom de la propriété et valeur représente sa valeur. Il ne doit pas y avoir d'espace entre l'option et la définition ni même dans la définition. Il faut utiliser autant d'option -D que de propriétés à définir.
-classpath chemins ou -cp chemins	permet d'indiquer les chemins de recherche des classes nécessaires à l'exécution. Chaque répertoire doit être séparé avec un point virgule. Cette option annule l'utilisation de la variable système CLASSPATH
-classic	Permet de préciser que c'est la machine virtuelle classique qui doit être utilisée. Par défaut, c'est la machine virtuelle utilisant la technologie HotSpot qui est utilisée.
-version	Affiche des informations sur l'interpréteur
-verbose ou -v	Permet d'afficher chaque classe chargée par l'interpréteur
-X	Permet de préciser des paramètres particuliers à l'interpréteur.

Le paramètre classe doit être un fichier .class dont il ne faut pas préciser l'extension. La classe contenue dans ce fichier doit obligatoirement contenir une méthode main.

- **Jar**

Le diminutif de Java ARchive. C'est un format de fichier qui permet de regrouper des fichiers contenant du bytecode Java (fichier .class) ou des données utilisées en tant que ressources (images, son, ...). Ce format est compatible avec le format ZIP : les fichiers contenus dans un jar sont compressés de façon indépendante du système d'exploitation.

6. Variables et Opérations

6.1. Types de variables

- **Variables numériques**

Type: byte (1 octet)
variable compris entre -128 et +127

Type: short (2 octets)
variable compris entre -32768 et + 32767

Type: int (4 octets)
variable compris entre -- 2147483648 et 2147483647

Type: long (8 octets)
variable compris entre -2^{31} et $2^{31}-1$

Type: float (4 octets)
on écrit un "f" à la fin du nombre.

Type: double (8 octets)
idem float avec plus de chiffre après la virgule mais sans le suffixe à la fin

Exemple :

```
byte vitesse;
vitesse=124;

short km;
km=130;

int cm;
cm=1300000;

float kilo=250.5f;

double gramme=25043;
```

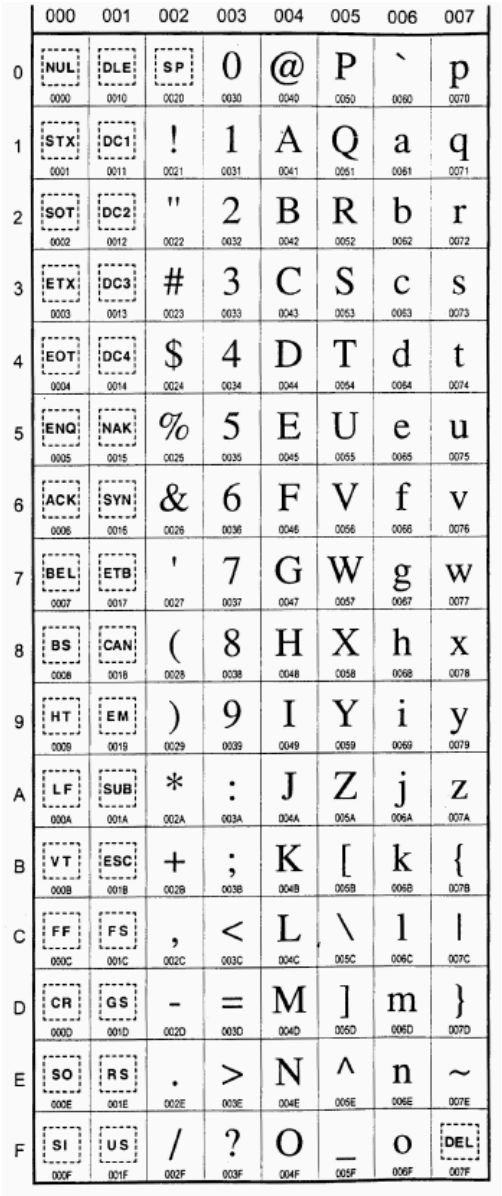
- **Variables stockant un caractère**

Type: boolean
ne peut contenir que 2 valeurs "true/false"

Type: char
un caractère stocké entre apostrophe

```
char carac;
carac = 'a';
```

Les caractères sont codés à partir de la table de codage Unicode.



	000	001	002	003	004	005	006	007
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070
1	STX 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071
2	SOT 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073
4	EOT 0004	DC4 0014	$ 0024	4 0034	D 0044	T 0054	d 0064	t 0074
5	ENQ 0005	NAK 0015	% 0025	5 0035	E 0045	U 0055	e 0065	u 0075
6	ACK 0006	SYN 0016	& 0026	6 0036	F 0046	V 0056	f 0066	v 0076
7	BEL 0007	ETB 0017	' 0027	7 0037	G 0047	W 0057	g 0067	w 0077
8	BS 0008	CAN 0018	(0028	8 0038	H 0048	X 0058	h 0068	x 0078
9	HT 0009	EM 0019	) 0029	9 0039	I 0049	Y 0059	i 0069	y 0079
A	LF 000A	SUB 001A	* 002A	: 003A	J 004A	Z 005A	j 006A	z 007A
B	VT 000B	ESC 001B	+ 002B	; 003B	K 004B	[005B	k 006B	{ 007B
C	FF 000C	FS 001C	, 002C	< 003C	L 004C	\ 005C	l 006C	| 007C
D	CR 000D	GS 001D	- 002D	= 003D	M 004D	] 005D	m 006D	} 007D
E	SO 000E	RS 001E	. 002E	> 003E	N 004E	^ 005E	n 006E	~ 007E
F	SI 000F	US 001F	/ 002F	? 003F	O 004F	_ 005F	o 006F	DEL 007F

Attention :

Pour un même nombre, ne pas confondre le caractère et la valeur du code Unicode auquel il fait référence.

Si on fait : char Var=9;

Ici on fait référence au caractère qui a la valeur Héxa « 9 », soit la « tabulation ».

Si on fait: char Var='9';

Ici on fait référence au caractère « 9 » qui a la valeur Héxa 57.

On pourrait également initialiser cet exemple directement comme cela : char Var=57;

- **Chaîne de caractères**

La chaîne de caractère en Java n'est pas considérée comme un type primitif mais comme une classe, fournie dans le package ***java.lang***.

*essai.java

```
public class essai {

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

        String chaine1= "Hello";
        String chaine2= new String ();
        chaine2="Hello1";
        String chaine3= new String ("Hello2");
    }

}
```

Commentaires :

Attention : La convention veut que String commence par une majuscule car celui-ci est un objet.

6.2. Les Constantes

Une constante est une donnée dont la valeur est inchangeable lors de l'exécution d'un programme. Cela ne s'applique qu'aux variables de type primitif.

```
final float TOURSUR2PI=6.2831f;
```

6.3. Les operateurs

Opérateurs arithmétiques	
Addition	+
Soustraction	-
Multiplication	*
Division	/
Modulo	%

Opérateurs sur les bits	
NON binaire	~
Décalage à droite	>>
Décalage à gauche	<<
ET binaire	&
Complément à 1	^
OU binaire	|

Opérateurs incrémentales	
Incrémente de 1	+
Décrémente de 1	-

Opérateurs logiques	
Négation	!
ET logique	&&
OU logique	||

Opérateurs relationnels	
Égal à	= =
Inférieur à	<
Supérieur à	>
Inférieur ou égal à	<=
Supérieur ou égal à	>=
Différent à	!=

6.4. Priorité des operateurs

++++++++++++++++	()	[]	.					
++++++++++++++++	--	++	!	~	- (un opérande)	+ (un opérande)	(casting)	new
+++++++++++++++	*	/	%					
+++++++++++++++	+	-						
+++++++++++++	<<	>>	>>>					
++++++++++++	<	<=	>=	>	instanceof			
++++++++++	==	!=						
+++++++++	&							
++++++++	^							
+++++++	|							
++++++	&&							
+++++	||							
++++	?	:						
+++	=	+=	-=	*=	/=	%=	&=	
++	<<=	>>=	>>>=	^=	|=			
+	,							

7. Fonction d'affichage et de saisie

7.1. Affichage sur une console

Pour afficher du texte ou la valeur d une variable, on utilise l'instruction *System.out.println (variable)* ou *System.out.println (variable)*.

```
public class essai {

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

        int a=11,b=3;
        System.out.println(a);
        System.out.print("la valeur de b:"+b);
        System.out.print("A \n suivre");
    }

}
```

Commentaires :

L'opérateur "+" permet de concaténer l'information et ainsi permet d'économiser des variables (mémoire).

Tableau récapitulatif des différentes options de System.out.print

Code de contrôle	Signification
\n	Nouvelle ligne
\b	Espace arrière
\t	Tabulation
\f	Saut de page
\r	Retour chariot
\'	Apostrophe
\\	Antislash
\"	Guillemet

7.2. La Classe Scanner

Pour que JAVA puisse lire ce que vous tapez sur le clavier, on utilise l'objet *Scanner.* Ci-dessous la syntaxe de *Scanner* :

```
Scanner saisie= new Scanner (System.in);
```

Voici un exemple d'instructions pour récupérer une chaîne de caractère et un entier saisie au clavier.

```
import java.util.Scanner;
public class essai {

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

        System.out.println("Nom de famille:");
        Scanner saisie= new Scanner (System.in);
        String message=saisie.nextLine();
        System.out.println("Age:");
        int age=saisie.nextInt();
        System.out.println(message + "\n" + age);

    }

}
```

Commentaires :

Scanner fait partie du package *java.util,* il faut impérativement importer ce package pour éviter une erreur sur l'IDE.

8. Conversion de Type (Cast)

On peut changer le type d'une variable vers un autre type en utilisant l'opérateur cast : (type) en le mettant devant l'identificateur de la variable à convertir.

```
import java.util.Scanner;
public class essai {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int entier1=4,entier2=3;
        float reel;
        reel=(float)entier1/(float)entier2;
        System.out.printf("le resultat="+ reel);
    }

}
```

Commentaires :

9. Structure de contrôle

9.1. Conditions

- **If**

```
if (condition) instruction ;

            ou

if (condition)
{
        instruction 1 ;
        …
        instruction n ;
}
```

- **If … Else**

```
if (condition) instruction1 ; else instruction2;

                    ou

if (condition)
{
        instruction 1 ;        //bloc 1
        …
        instruction n ;
}
else
{
        instruction 1 ;        //bloc 2
        …
        instruction n ;
}
```

- **If … Else if … Else**

```
if(condition1) instruction 1;else if(condition2) instruction 2;…;else(condition n) instruction n;

                                        ou

if (condition1)
{
        instruction 1 ;        //bloc 1
        …
}
else if (condition2)
{
        instruction 2 ;        //bloc 2
        …
}
   …
else  (condition n)
{
        instruction n ;        //bloc n
        …
}
```

9.2. Les Boucles

- **While**

```
while (condition) instruction ;

            ou

while (condition)
{
        instruction1 ;
        …
        instruction n ;
}
```

- **Do … While**

```
do instruction ; while (condition) ;

            ou

do
{
        instruction1 ;
        …
        instruction n ;
}
while (condition) ;
```

- **For**

```
for(expr1; expr2; expr3)
{
        instruction1 ;
        …
        instruction n ;
}
```

expr1: déclaration et initialisation de la variable

expr2:condition sur cette variable

expr3 :incrémentation ou décrémentation de la variable

9.3. Choix

- **Switch**

```
switch (variable)
                {
        case constante1 : instruction1 ;
        …
        case constante n : instruction n ;
        default : instruction n+1 ;
}
```

TD n°1 : Comptage de bouteilles

Cf. énoncé

10. Les Tableaux

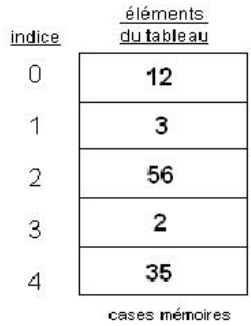
10.1. Tableau à une dimension

Un tableau est une suite de cellules consécutives en mémoire contenant des données de même type.
La déclaration consiste à définir un type de données et une taille.
Un indice permet de spécifier l'adresse relative de chaque cellule par rapport au début du tableau. Cet indice doit être un entier. La première cellule aura l'indice 0, la seconde l'indice 1, …, la nième (le dernier) l'indice n-1.

Syntaxe:

<type du tableau> <nom du tableau> [ ] = {contenu du tableau} ;

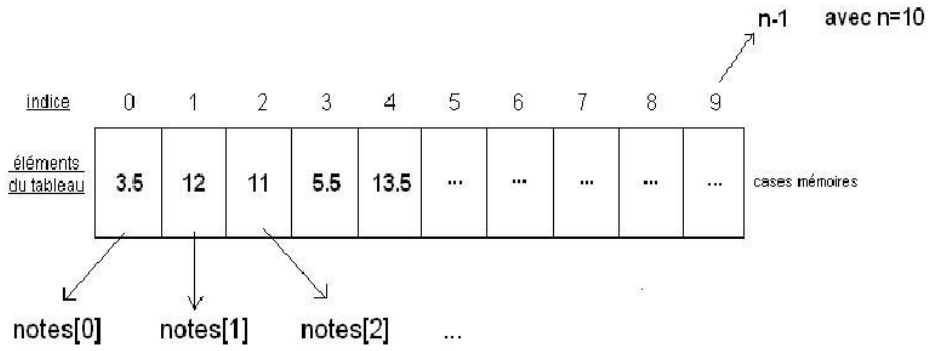
int *tab [ ] = {1,2,3,4,5};* /* Déclare un tableau de 5 entiers déclarés et initialisés */



- Premier élément du tableau => indice 0

- Dernier élément du tableau => indice (n-1)
(avec n : nb max éléments du tableau)

float *notes[ ] =****new float****[10];* /* Déclare un tableau de 50 réels initialisés à 0 */



10.2. Tableau à 2 dimensions

Le nombre de dimension n'est pas limité.
Syntaxe :

<type du tableau> <nom du tableau> [ ] [ ] = {{1ière colonne} , …, {nième col}};

On peut se servir d'un tableau avec plusieurs dimensions pour faire des matrices par exemple.

10.3. Initialisation de tableau

On peut initialiser les tableaux au moment où après leur déclaration.

Exemple : **float** nombre[ ] = {2.67f,5.98f,-8f,0.09f}; //Au moment de la déclaration

nombre[1] =5.98; //Après la déclaration

Application :

- Afficher les éléments du tableau nombre[ ].
- Initialiser toutes les cases mémoire du tableau nombre[ ] à 999,99.
- Afficher les éléments du tableau.

10.4. Remplissage de tableau

Le plus simple est l'affectation élément par élément via l'utilisation d'une boucle.

Application :

- Déclarer le tableau nombre[ ] comme précédemment puis un nouveau tableau bis[ ]
- Afficher les éléments du tableau nombre[ ] et bis[ ].
- Initialiser toutes les cases mémoire du tableau bis[ ] à nombre[ ].
- Afficher les éléments des 2 tableaux.

Les Méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe. Si un bout de code se répète alors on peut écrire une méthode.

Avantages : simplicité et clarté du code.

1. Syntaxe d'une méthode

```
modificateurs type_retourné nom_méthode ( type_arg1, ... ) {
        // définition des variables locales et du bloc d'instructions
        ...
}
```

Nom_méthode:
nom par laquelle on pourra appeler la méthode correspondante. Par convention on choisit un nom résumant la fonction de celle-ci.

Type_arg*:
Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis lors de l'appel.

Modificateurs :

Modificateur	Rôle
public	La méthode est accessible aux méthodes des autres classes
private	L'usage de la méthode est réservé aux autres méthodes de la même classe
protected	La méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous classes
final	La méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	La méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	La méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	Le code source de la méthode est écrit dans un autre langage

Type_retourné :
Peut-être élémentaire ou correspondre à un objet. La valeur de retour de la méthode doit être transmise par l'instruction return. Si la méthode ne retourne rien, alors on utilise **void**.

Exemple d'une méthode sans valeur retournée

```
public class essai1 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		/*Déclaration des variables entieres*/
		int a=0;
		System.out.println("Donner les valeurs de a: ");
		Scanner sais=new Scanner(System.in);
		a=sais.nextInt();
	    traitement(a);
	    System.out.print(a);
	}

	/*Déclaration des methodes*/
	public static void traitement( int b){
		System.out.println("\t\tJe traite");
		/*traitement*/
		b*=b;
	}
}
```

Observations :

Exemple d'une méthode avec valeur retournée

```
public class essai2 {
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		System.out.print("\tLa val au cube vaut :"+cube(saisie()));

	}
	/*Methodes*/
	public static float saisie()
	{
		float val;
		Scanner sais=new Scanner(System.in);
		System.out.print("Entrer une valeur: ");
		val=sais.nextFloat();
		return val;
	}
	public static float cube(float a)
	{
		int i;
		float cub=1;
		for( i=1 ; i<=3 ; i++) cub*=a;
		return cub;
	}
}
```

Observations :

TD n°2 : Nombre mystère

Cf. énoncé

Les classes et les objets en Java

1. Les classes et les objets en Java

1.1. Déclaration et définition d'une classe en java

Une classe en java sera toujours du type public. Une méthode **sera toujours déclarée et définie** à l'intérieur d'une classe.

```
Public class Point {
private int x ;
private int y ;
public A (…..) {…….}  // déclaration et définition d'un constructeur
public double calcul(…….) {…….}// déclaration et définition d'une méthode
protected void affiche(…..) {…….}//
}
```

Un membre d'une classe peut être initialisé lors de sa déclaration dans la classe.

```
Public class Point {
private int x=2 ;
private int y=3 ;
public A (…..) {…….}  // déclaration et définition d'un constructeur
public double calcul(…….) {…….}// déclaration et définition d'une méthode
protected void affiche(…..) {…….}//
}
```

1.2. Création d'un objet

A l'intérieur d'une méthode quelconque, une déclaration reste toujours le nom de la classe suivi du nom de l'objet :

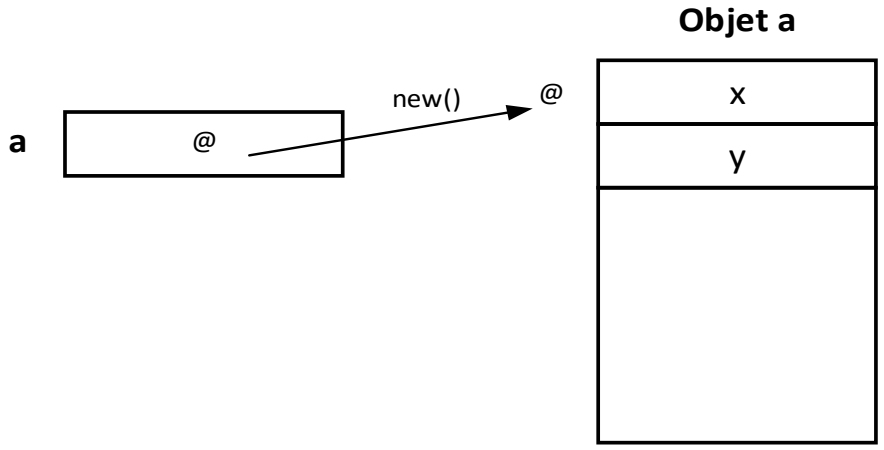
```
Point a ; //syntaxe correcte.
```

Cependant, contrairement à la déclaration d'une variable d'un type primitif (comme int n), on ne réserve pas d'emplacement pour un objet de type Point, mais seulement un emplacement pour une référence à un objet de type Point. L'emplacement pour l'objet proprement dit sera alloué sur une demande explicite du programme, en faisant appel à un opérateur nommé new. Ainsi, l'expression :

new Point() ; // Attention à la présence des parenthèses

Crée un emplacement pour un objet de type Point et fournit sa référence en résultat. Par exemple, on pourra procéder à cette affectation :

```
a= new Point() ;
```



Cette déclaration crée une case mémoire, nommée **a**, destinée à contenir une référence vers l'adresse où sont stockées les informations concernant l'objet **a**.

L'opérateur **new** va permettre de réserver l'espace mémoire pour l'objet **a.**

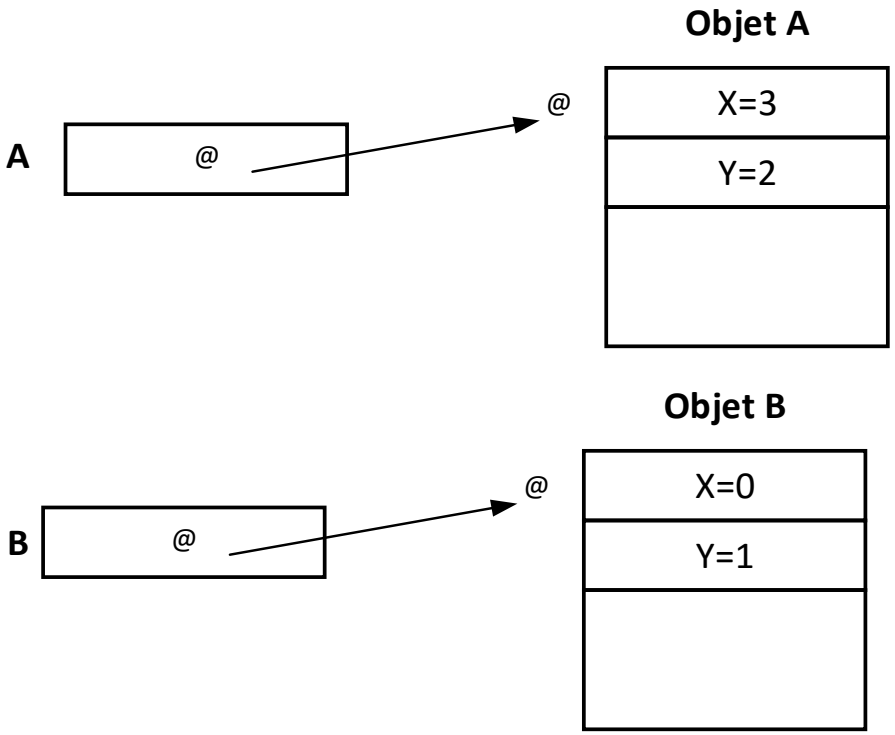
1.3. La notion de référence en Java

Il existe des variables de classe, destinées à contenir des références sur des objets. On peut affecter des objets avec d'autres objets :

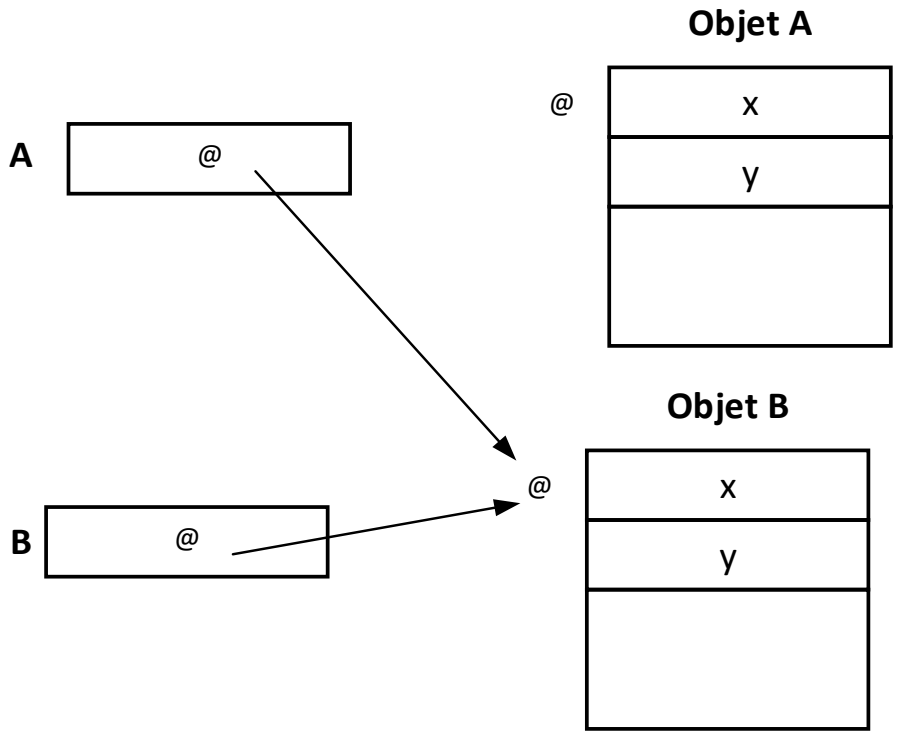
Point A,B ; // Variables de classe

*A= **new** Point (3,2) ; // On réserve une case mémoire pour A qui contient une référence*

*B= **new** Point (0,1) ; // On réserve une case mémoire pour B qui contient une référence*



Si A=B



L'objet **A** a donc la même référence que l'objet **B**. La modification d'un membre de **A**, **X** ou **Y** impactera indirectement avec la même valeur les membres **X** et **Y** de l'objet **B**.

De même la comparaison de **A** et **B** (**A==B**) ne veut pas dire :

Est que les objets A et B ont les mêmes valeurs

Mais est ce qu'ils ont la même référence ?

Lorsqu'on passe en paramètre d'entrée à une méthode une variable du type primitif (**char**, **int** ….) on fait un passage par copie de ce type.

```
public class point {

private int X ;
public void setX (int a) {
X=a ;
}
……..……
……..……

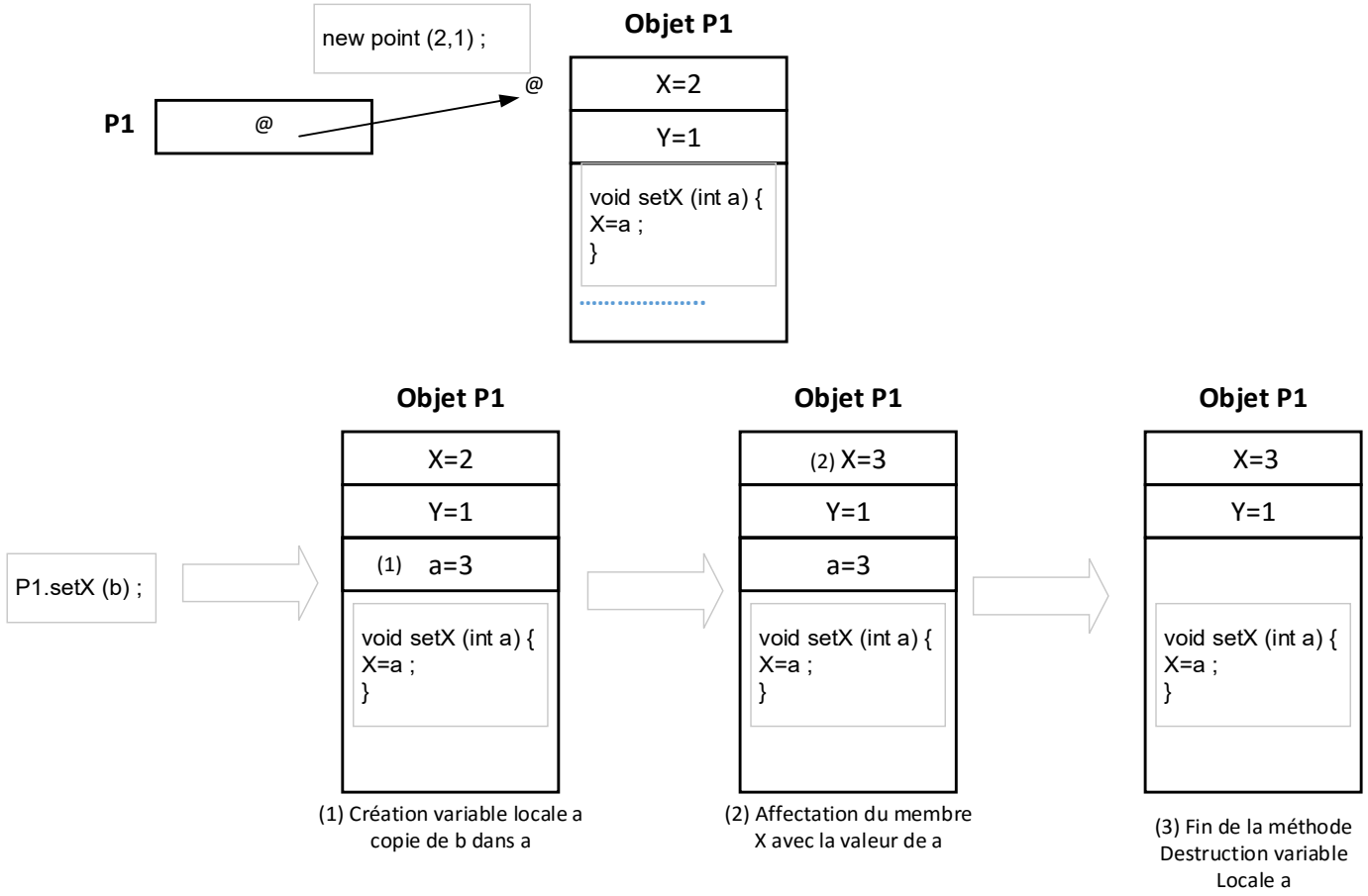
}

public static void main(String[] args) {

int b=3;
point P1 = new point (2,1) ;

P1.setX (b) ;

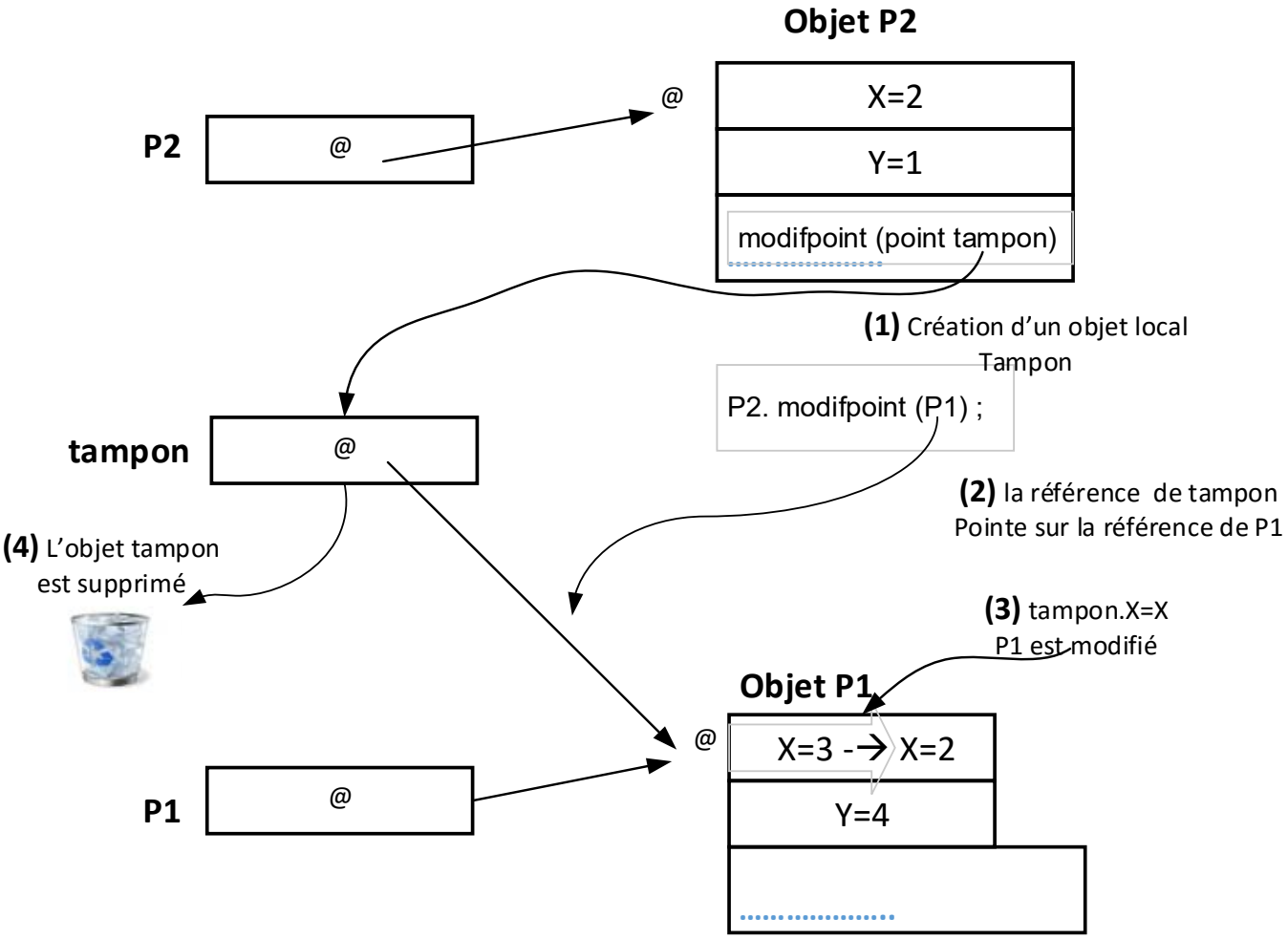
}
```



Lorsque le passage de paramètre d'entrée est un objet, l'objet étant défini par son adresse (référence), la valeur passée en paramètre n'est plus la valeur réelle de la variable mais l'adresse de l'objet.
Grâce à cela, les modifications apportées sur l'objet passé en paramètre et réalisées à l'intérieur de la méthode sont visibles en dehors même de la méthode.

```
public class point {
private int X ;
void double modifpoint (point tampon) {
tampon.X=X ;
}
……..
……..
}
```

```
public static void main(String[] args) {
private P1 = new point (2,1) ;
private P2= new point (3,4) ;
P2. modifpoint (P1) ;
}
```



Un objet peut être créé par un autre objet, par une méthode ou la méthode main qui correspond au point de départ de notre programme.

```
public class TestA{
public static void main (String args[]){
A obj1= new A() ; // création de l'objet obj1 avec le constructeur par défaut
}
 }
```

1.4. Particularité des méthodes ou des membres du type static et final

1.4.1. Le mot clé static

Lorsque l'interpréteur rencontre le mot-clé static devant un membre ou une méthode, il réserve un seul et unique emplacement mémoire pour y stocker la valeur ou le pseudocode associé. Cet espace mémoire est communément accessible pour tous les objets du même type.
Lorsque le mot-clé static n'apparaît pas, l'interpréteur réserve, à chaque appel de l'opérateur new, un espace mémoire pour y charger les données décrites dans la classe.
Si vous définissez un champ comme static, il ne peut en exister qu'un seul par classe.

Grâce au mot-clé static, la variable de classe nombre est un espace mémoire commun, accessible par tous les objets créés.

```
public class Math
{
. . .
public static final double PI = 3.14159265358979323846;
. . .
}
```

Vous pouvez accéder à cette constante dans vos programmes à l'aide de Math.PI.

Vous pouvez définir un champ d'instance comme final. Un tel champ doit être initialisé lorsque l'objet est construit. C'est-à-dire qu'il doit être certain que la valeur du champ est définie après la fin de chaque constructeur**. Ensuite, il ne peut plus être modifié**.

1.5. La surcharge des méthodes

La surcharge s'effectue si plusieurs méthodes possèdent le même nom mais avec des paramètres d'entrées différentes. Le compilateur Java se charge de déterminer laquelle il va employer.

```
public class mathematique {

public int minimum (int x, int y) {
return  x<=y ? x :y ;}

public double minimum (double x, double y) {
return  x<=y ? x :y ;}


}
```

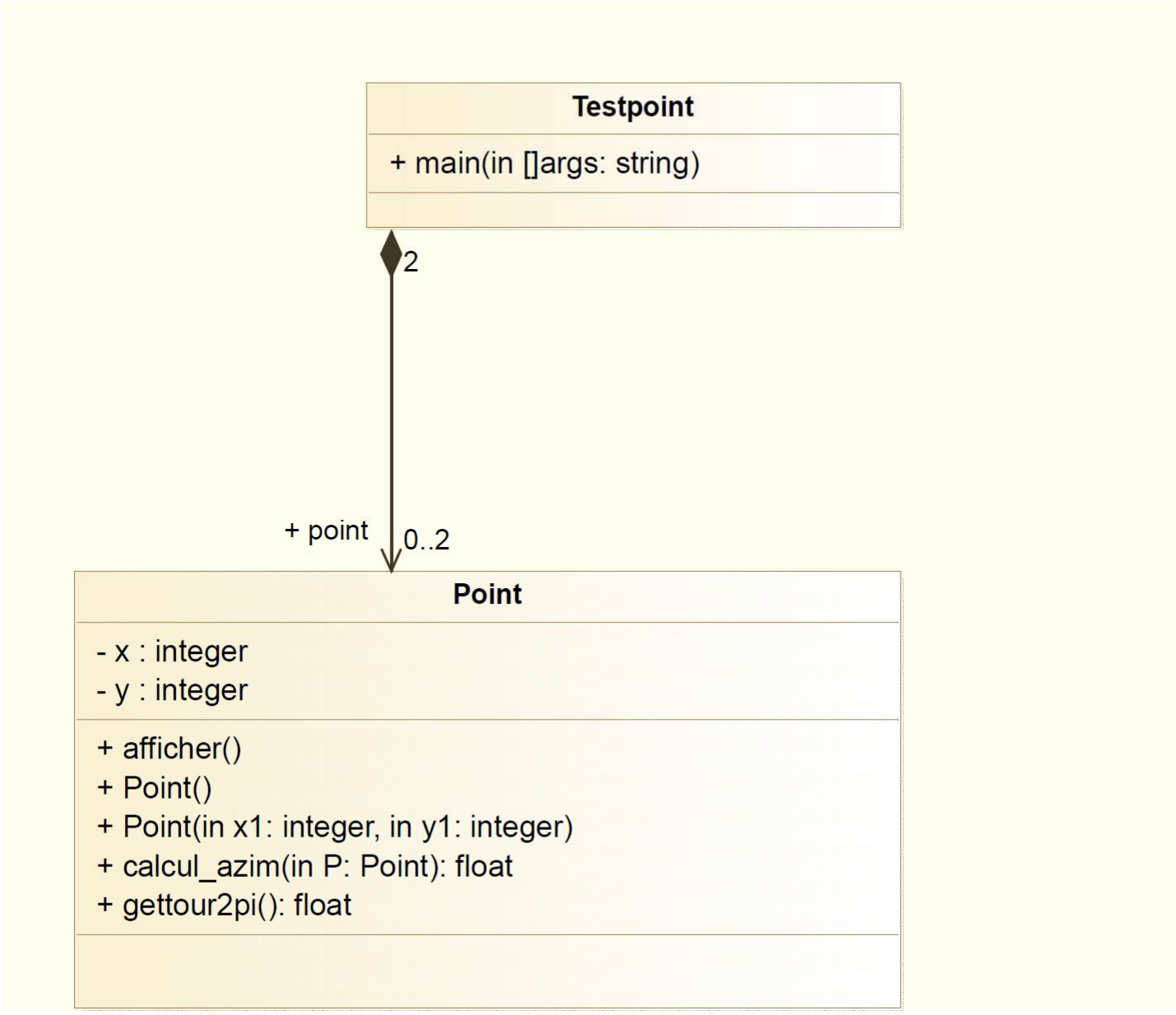
2. La variable This

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. **this** est un objet qui est égale à l'instance de l'objet dans lequel il est utilisé.

Exemple : Création d'un objet en JAVA

Soit la classe point, sous eclipse, déclarer et définir la méthode calcul_azim(point P), qui permet de calculer l'azimuth entre l'objet Point qui a instancié la méthode et le Point P passé en paramètre.
Compléter la méthode main et la classe Testpoint pour tester la méthode calcul_azim

Le diagramme de classe est le suivant :



Code source :

Classe Point

```
import java.util.Scanner;
import  java.lang.Math.*;

public class Point {
      private int x;
      private int y;
      public static final float TOURSUR2PI=6.2831f;

      public void afficher ()
      {
            System.out.println(" je suis un pt 1 de coord ordonnées " +x) ;
            System.out.println(" je suis un pt 1 de coord abcisses " +y) ;

      }


public static float gettour2pi()
      {
            return TOURSUR2PI;
      }

      public  Point()
      {
            System.out.println(" veuillez saisir x1  \n" ) ;
            Scanner lectureClavier = new Scanner(System.in);
            x=lectureClavier.nextInt();
            System.out.println(" veuillez saisir y1  \n" ) ;
            lectureClavier = new Scanner(System.in);
            y=lectureClavier.nextInt();

      }

      public  Point(int x1, int y1)
      {
            this.x=x1;
            this.y=y1;

      }
```

```
public float calcul_azim(Point P)
      {  float result=0;
```

```
      return result;
      }
}
```

Classe Testpoint

```
import java.util.*;

public class Testpoint {

      /**
       * @param args
       */
      public static void main(String[] args) {
            // TODO Auto-generated method stub
            float res=0;
            Point A=new Point();// Point A avec constructeur par défaut
            Point B=new Point(3,4);// Point B constructeur x=3, y=4
            A.afficher();
            B.afficher();
```

```
      }

}
```

TD n°3 : Jeux sur les caractères

Cf. énoncé

Initialisation d'un objet et constructeur

1. Initialisation d'un objet

On a vu précédemment qu'il était possible d'initialiser les membres d'un objet directement dans la classe. Cependant il peut être nécessaire de recourir à une ou des méthodes constructeur pour attribuer des valeurs aux membres ou attributs d'un objet.

La définition d'une classe peut se présenter ainsi :

```
public class Point {  // constructeur
private int x ;
private int y ;
public Point (int abs, int ord)
{
X=abs ;
Y=ord ;
}
public void deplace (int dx, int dy)
{
X=x+dx ;
Y=y+dy
}
public void affiche()
{
system.out.println(" je suis un pt de coord " +x+ " "+y) ;
}

}
```

Si on ne déclare aucun constructeur dans la classe, lors de la création de l'objet (opérateur new), le compilateur utilise un constructeur par défaut par défaut **les membres d'un objet (attributs) sont toujours initialisés par défaut :**

Boolean	**False**
char	**caractère de code nul**
entier (byte, short, int, long)	**0**
flottant (float, double)	**0.**
objet	**Null**

Lorsqu'un constructeur par défaut (aucun paramètre d'entrée) ou non est défini par le programmeur dans la classe, le constructeur proposé par défaut par défaut par le compilateur Java n'existe plus.

Point p1= new Point() ;

Comme en C++, le langage Java permet la définition de plusieurs constructeurs, ou méthodes, à l'intérieur d'une même classe, du fait que la construction des objets peut se réaliser de différentes façons. Lorsqu'il existe plusieurs constructeurs, on dit que le constructeur est **surchargé**.

Les Exceptions

Lors de l'écriture d'un programme, la prise en compte d'erreurs prend une place très importante si l'on souhaite écrire un programme robuste. Java, propose un mécanisme de prise en compte des erreurs, fondé sur la notion d'exception. Une exception est un objet qui peut être émis par une méthode si un événement d'ordre "exceptionnel" (les erreurs rentrent dans cette catégorie) se produit. Un ensemble de trois mots clés permettent de détecter et de traiter ces erreurs : **try**, **catch** et **finally** , ainsi que : **throw** et **throws** qui permettent de les lever ou les propager.

Exemple:

```
import java.util.Scanner;
public class ExempleExcep {

    public static String month(int mois){
        if ((mois < 1) || (mois > 12)) {
            if (mois == 1)return "Janvier" ;
            else if (mois == 2)return "Février" ;
            else if (mois == 3)return "Mars" ;
            else if (mois == 4)return "Avril" ;
            else if (mois == 5)return "Mai" ;
            else if (mois == 6)return "Juin" ;
            else if (mois == 7)return "Juillet" ;
            else if (mois == 8)return "Aout" ;
            else if (mois == 9)return "Septembre" ;
            else if (mois == 10)return "Octobre" ;
            else if (mois == 11)return "Novembre" ;
            else return "Décembre" ;
        }
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.print("Entrez le numero d'un mois : ");
        Scanner scan = new Scanner(System.in);
        String choix = scan.nextLine();
        int numero = Integer.parseInt(choix);
        System.out.println(month(numero));
        System.out.println("Vivement le mois prochain !!!");
    }
```

Observations :

1. Classes d'exception

Une classe est considérée comme une classe d'exception dès lors qu'elle hérite de la classe Throwable . Un grand nombre de classes d'exception sont proposées dans l'API pour couvrir les catégories d'erreurs les plus fréquentes. Les classes d'exception les plus fréquemment utilisées sont récapitulées dans le tableau ci-dessous:

Classe d'exception	Description
NullPointerException	Se produit lorsqu'un pointeur *null* est reçu par une méthode n'acceptant pas cette valeur, ou lorsque l'on appelle une méthode ou une variable à partir d'un pointeur *null*.
IndexOutOfBoundsException	Se produit lorsque l'on essaie d'accéder à un élément inexistant dans un ensemble.
IOException	Les exceptions de cette classe peuvent se produire lors d'opérations d'entrées/sorties.
FileNotFoundException	Signale une tentative d'ouverture d'un fichier inexistant.
ClassCastException	Signale une erreur lors de la conversion d'un objet en une classe incompatible avec sa vraie classe.
AWTException	Les exceptions de cette classe peuvent se produire lors d'opérations de type graphique.

2. Try, Catch et Finally

Il faut protéger le bloc du code qui pourrait lever une exception à l'intérieur d'un bloc **try**, et traiter l'exception à l'intérieur d'un bloc **catch**.
Si l'exécution du code "sous surveillance" se passe correctement, le bloc **catch** n'est pas exécuté.

Exemple :

```
public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.print("Entrez le numero d'un mois : ");

        try {
                Scanner scan = new Scanner(System.in);
                String choix = scan.nextLine();
                int numero=Integer.parseInt(choix);
                System.out.println(month(numero));
        }
        catch (IndexOutOfBoundsException e) {
                System.err.println("Numero incorrect : "+ e.getMessage());

        }
        catch (NumberFormatException e) {
                System.err.println("Entrée incorrecte : "+e.getMessage());
                //e.printStackTrace();

        }
        catch (IOException e) {
                System.err.println("Erreur d'accès : "+ e.getMessage());

        }

        finally{
                System.out.print("\nToujours executé");
        }
        System.out.print("\nVivement le mois prochain !!!");
    }
}
```

Observations :

Lorsqu'un code est susceptible de provoquer différentes exceptions non reliées par l'héritage, on utilise plusieurs clauses **catch** pour une clause **try.** Il faut évidemment placer les sous-classes d'exceptions avant leur classe parente. Un seul bloc sera exécuté.

Finally: (optionnel) Cette clause est réservée pour les actions qui doivent être exécutées qu'il y ait exception ou pas.

3. Throws, Throw

Throws: On ajouter le mot clé **throws** dans la signature de la méthode si cette méthode peut lever des exceptions. L'exception levée devra être traitée dans une clause **catch**.

Throw: Permet de lever une exception manuellement en instanciant un objet de type Exception (ou un objet hérité).

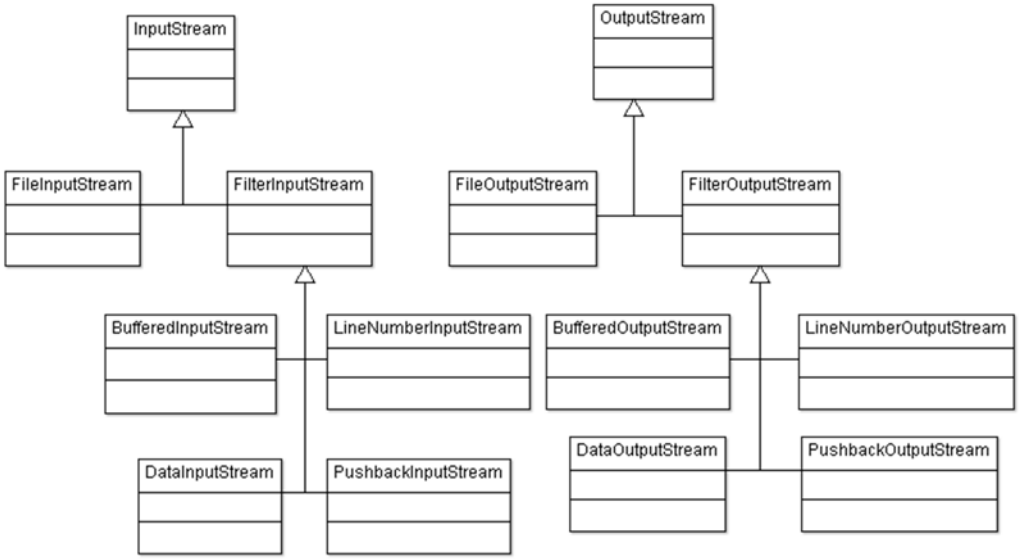
Exemple :

```
public static String month(int mois)throws IndexOutOfBoundsException,IOException
        {
                if ((mois < 1) || (mois > 12)) {
                        throw new IndexOutOfBoundsException("le numero du mois qui est "+ mois+ " doit être compris entre 1 et
                        12");
                }
                        if (mois == 1)return "Janvier";
                        else if (mois == 2)return "Février";
                        else if (mois == 3)return "Mars";
                        else if (mois == 4)return "Avril";
                        else if (mois == 5)return "Mai";
                        else if (mois == 6)return "Juin";
                        else if (mois == 7)return "Juillet";
                        else if (mois == 8)return "Aout";
                        else if (mois == 9)return "Septembre";
                        else if (mois == 10)return "Octobre";
                        else if (mois == 11)return "Novembre";
                        else return "Décembre";
                }
        }
```

Projet : saisiExcept

Les Fichiers

En Java, les entrées et les sorties d'informations se présentent comme des flux. L'information est transférée sous forme d'octets à partir ou vers une source qui peut être variée : un fichier, un périphérique, une connexion sur un autre ordinateur.
Il existe différents objets plus ou moins faciles à utiliser. En fonction de l'utilisation qu'on vaut faire on utilisera tel ou tel objet. Ces objets font partie de la hiérarchie suivante:



*Attention : Les flux d'entrée **in** pour la lecture de flux et les flux de sorties **out** pour l'écriture de flux.*

1. Les Exceptions

Si on essaie d'ouvrir un fichier qui n'existe pas, il y a déclenchement d'une exception de type **FileNotFoundException**.
Si on atteint la fin d'un fichier, il y a lancement d'une exception de type **EOFException.**

2. File

Pour pouvoir utiliser un objet de type File, vous devez importer le package java.io.File avec l'instruction suivante :

import java.io.File;

Syntaxe :

File *<nom_de_l'objet>* **= *new File(****"nom_du_fichier"****);***

Méthodes de l'objet File :

Méthodes	Fonction
fichier.getAbsolutePath()	Chemin absolu
fichier.getName()	Nom du fichier
fichier.exists()	true si le fichier existe, false sinon
fichier.isFile()	true s'il s'agit d'un fichier, false sinon
fichier.length()	Taille du fichier en octets
fichier.isDirectory()	true s'il s'agit d'un dossier, false sinon

Projet : fichier-class file

3. BufferedInputStream - BufferedOutputStream

Cette classe permet d'avoir un tampon à disposition dans la lecture du flux. Les données vont remplir le tampon, et dès que celui-ci est plein, le programme accède aux données. L'utilisation de cette classe accélère la lecture des fichiers.

Quelques méthodes :

BufferedInputStream	BufferedOutputStream
int read( ) This method reads the next byte of data from the input stream.	**void flush**() This method flushes this buffered output stream.
int read(**byte**[ ] b, **int** off, **int** len) This method reads bytes from this byte-input stream into the specified byte array, starting at the given offset.	**void write**(**byte**[] b, **int** off, **int** len) This method writes len bytes from the specified byte array starting at offset off to this buffered output stream.
void close() This method closes this input stream and releases any system resources associated with the stream.	**void write**(**int** b) This method writes the specified byte to this buffered output stream.

Projet : fileinput

4. RandomAccessFile

Permet de faire des lectures ou écritures à une position aléatoire dans un fichier.

Syntaxe :

RandomAccessFile <*nom_de_l'objet*> = ***new RandomAccessFile(***"*nom_du_fichier* ",***"flux d'accès");***

Le mode d'ouverture d'un flux à accès direct peut être :

r: Ouverture en lecture seule. L'appel d'une méthode d'écriture lève une exception **IOException**.
rw: Ouverture en lecture/écriture. Si le fichier n'existe pas, on essaie de le créer.
rws: Ouverture en lecture/écriture. La mise à jour du contenu du fichier et de ses méta-données est faite de façon synchrone.
rwd: Ouverture en lecture/écriture. La mise à jour du contenu du fichier est faite de façon synchrone.

Quelques méthodes :

Méthodes	Fonction
void seek(long pos)	Positionne le pointeur.
long length()	Retourne la longueur du fichier.
void close()	Fermeture du fichier
int read()	Lit un octet dans le fichier
int read(**byte**[ ] b)	Lit jusqu'à taille de b octets dans le fichier et les range dans b
int read(**byte**[] b, **int** d, **int** l)	Lit jusqu'à l octets dans le fichier et les range dans b à partir de d.
byte readByte()	Lit un octet.
char readChar()	Lit un caractère.
String readLine()	Lit une suite d'octets jusqu'à une marque de fin de ligne, ou la fin fu flux, et retourne ces octets sous forme de chaîne de caractères.
void write(**byte**[] b)	Ecrit le tableau d'octets b.
void write(**byte**[] b, **int** d, **int** l)	Ecrit l octets du tableau d'octets b, à partir de d.
void write(**int** b)	Ecrit l'octet de droite contenu dans b.
void writeByte(**int** v)	Ecrit l'octet v.
void writeChar(**int** v)	Ecrit le caractère contenu dans v.
void writeInt(**int** v)	Ecrit l'entier v.
void writeBytes(**String** s)	Ecrit la chaîne de caractères sous la forme d'une suite d'octets. Seul l'octet de droite de chaque caractère est écrit.
void writeChars(**String** s)	Ecrit la chaîne de caractères.

TD n°4 : Cryptographie

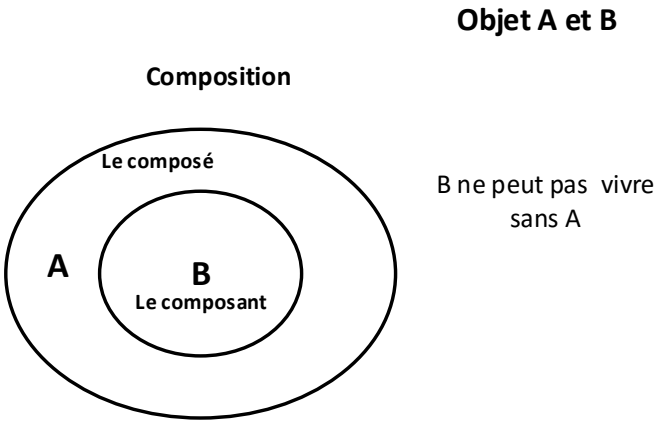
Cf. énoncé

Notion sur la composition et l'agrégation en java

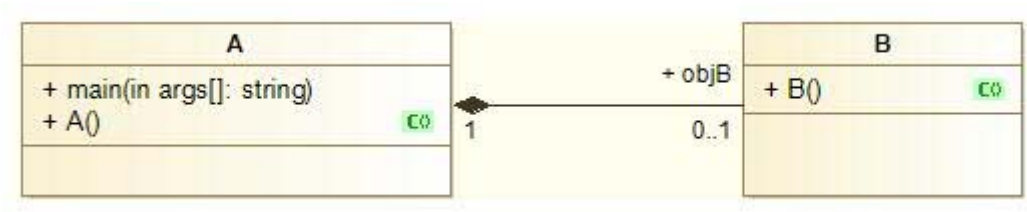
Les objets ont besoin de communiquer ensemble en s'envoyant des messages. Ces messages ne sont rien d'autres que des méthodes qui sont appelés.
Les relations qui lient les classes et par la suite les objets sont importantes pour comprendre l'interaction entre les différents objets. La notion de composition et d'agrégation vue en UML peuvent être implémentée en langage JAVA.
Nous allons tout d'abord voir la notion de composition dans un contexte JAVA.

1. La composition

La composition consiste à utiliser un objet comme attribut d'un autre objet



1.1. La composition unidirectionnelle




```
public class A {
  B objb;
  private int k=2;

  public A ()
  {
    objb=new B();// l'objet B ne peut
    exister sans l'objet A
  }

  public static void main(String[]
args)
  {
    A obja= new A();
  }
}
```

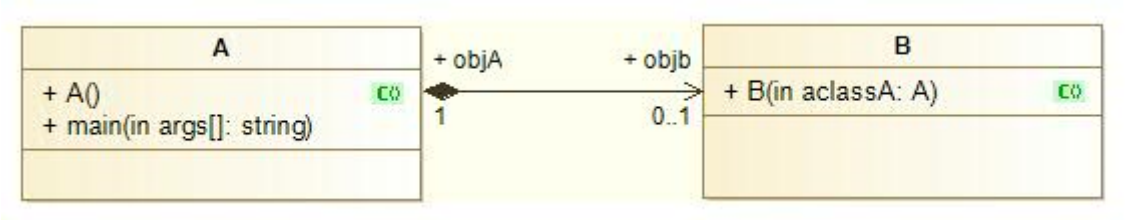
```
public class B {
public B ()
{
  System.out.println("l'objet B est bien créé
  par A");
  System.out.println("c'est une composition");
  System.out.println("l'objet A est le composé
  de B");
  System.out.println("l'objet B est le
  composant de A");
}

}
```

Le losange plein est toujours du côté du composé
objB sera connu à travers de A comme étant un objet de la classe B

Aucun objet de A ne sera connu de B

1.2. La composition bidirectionnelle



objA sera connu à travers de B comme étant un objet de la classe A
objB sera connu à travers de A comme étant un objet de la classe B

La bidirectionnalité est indiquée par une flèche (non obligatoire). Le fait d'avoir noté deux objets de chaque coté **objA** et **objB** signifie que l'on a une bidirectionnalité

```
public class A {

static  B objb;
int k=2;

public A ()
{
objb=new B();// l'objet B ne peut exister
sans l'objet A
}

      public static void main(String[]
args) {
A obja= new A();
objb.setA(obja);}}
```

```
public class B {

A obja;

public void setA(A aclasseA)
{
this.obja= aclasseA;
}

public B ()
      {
            // lien vers l'objet a de la
classe A
System.out.println("l'objet B est bien
créé par A");
System.out.println("c'est une
composition");
System.out.println("l'objet A est le
composé de B");
System.out.println("l'objet B est le
composant de A");
System.out.println("le composant (l'objet
B) connait le composé celui qui lui a
donné la vie (objet A)");
System.out.println("Si l'objet A appelle
la méthode setA de l'objet B");
      }
}
```

La bidirectionnalité peut aussi se traduire en utilisant le constructeur de la classe B

```
public class A {

B objb;
int k=2;

public A ()
      {
objb=new B(this);// l'objet B ne peut
exister sans l'objet A
      }


public static void main(String[] args) {
A obja= new A();
      }

}
```

```
public class B {

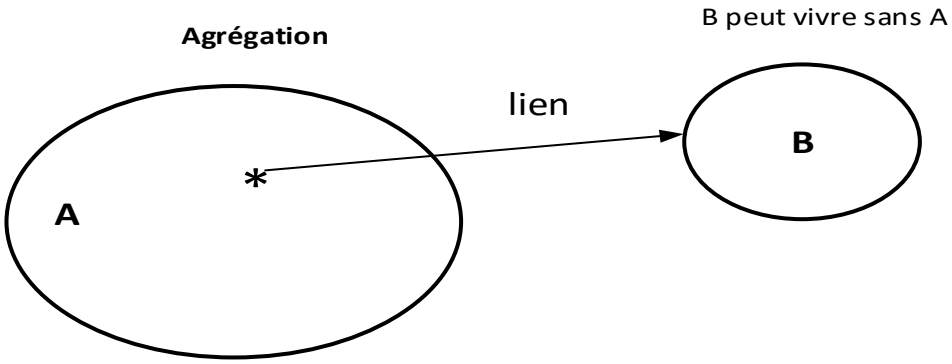
A a;
public B (A aclassA)
      {
this.a=aclassA;// lien vers l'objet a de
la classe A
System.out.println("l'objet B est bien
créé par A");
System.out.println("c'est une
composition");
System.out.println("l'objet A est le
composé de B");
System.out.println("l'objet B est le
composant de A");
System.out.println("le composant (l'objet
B) connait le composé celui qui lui a
donné la vie (objet A)");

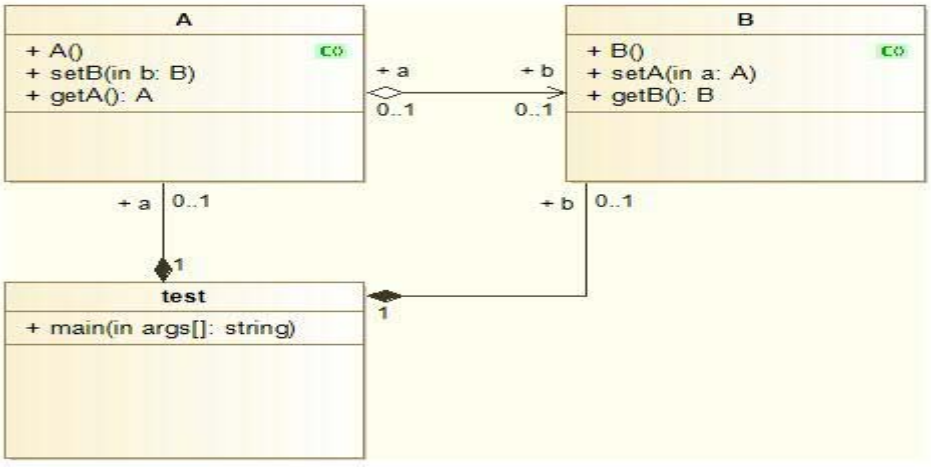
      }

}
```

1.3. L'agrégation

B n'est pas forcément créé par A mais A va faire un lien vers B





L'agrégation est représentée par un losange vide

```
public class A
{
      int k=0;// si pas de portée par défaut public

protected B b;
public void setB(B b)
{
this.b=b;
b.setA(this);// agrégation bidirectionnelle
}
public A getA()
{
return this;
}

public A()
{

}

}

public class B
{
protected A a;
public B(){
}
public void setA(A a)
{
this.a=a;
a.setB(this);// agregation bidirectionnelle
}
public B getB()
{
return this;
}
}

public class test {


      public static void main(String[] args) {
  A obja = new A();// obja peut exister sans objb
 B objb= new B();//objb peut exister sans obja
 obja.setB(objb);// obja fait un lien vers objb et objb fait un lien vers obja
  B objb1= new B();
 objb1=objb.getB();// équivant à faire objb1=objb;
 obja.k=3;
 System.out.println("l'objet obja fait un lien vers l'objet objb");

      }

}
```

Si A instancie B c'est une composition
Si c'est quelqu'un d'autre qui instancie B c'est une agrégation

L'héritage

1. Principe

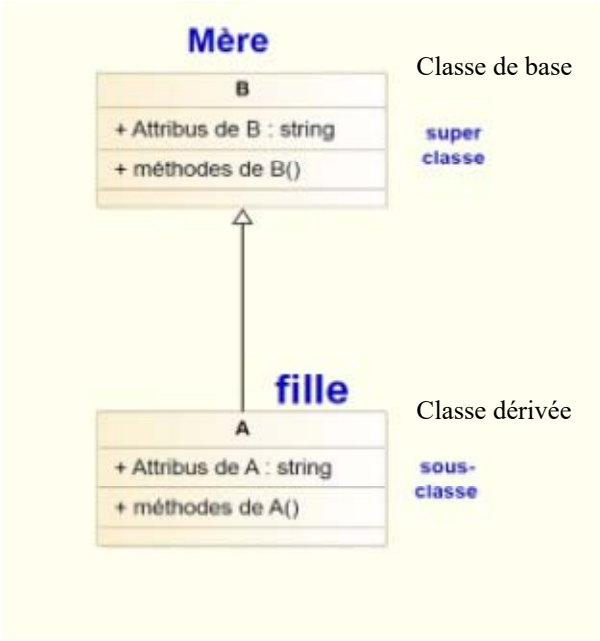
L'héritage est un mécanisme qui permet à une classe d'hériter de l'ensemble du comportement et des attributs ou membres d'une autre classe.

Grâce à l'héritage, une classe peut disposer immédiatement de toutes les fonctionnalités d'une classe existante. De ce fait, pour créer la nouvelle classe, il suffit d'indiquer dans quelle mesure elle diffère de la classe existante.

Une classe qui hérite d'une autre classe est appelée **sous-classe** classe fille ou classe dérivée, et la classe qui offre son héritage à une autre est appelée **super-classe**, classe mère ou classe de base.

La déclaration de l'héritage s'effectue de la manière suivante :

En C++	**En JAVA**
public class A : public B {	**public class A extends B {**
Attributs de A …….. …….	**Attributs de A** **……** **…….**
Méthodes de A …….. ……..	**Méthodes de A** **……..** **……..**
}	**}**



Le mot clé **extend** (notion d'héritage en java) remplace les deux points du C++. La notion d'encapsulation reste identique au C++, à la simple différence, une classe en java sera toujours du type publique.

Attributs ou méthodes de la classe mère	accessibilité directe par un objet de la classe fille	Portée dans les méthodes de la classe fille	accessibilité directe par un objet de la classe mère	Portée dans les méthodes de la classe mère
public	oui	oui	oui	oui
private	non	non	non	oui
protected	oui	oui	oui	oui

Les modificateurs de visibilité en Java



Visibilité des champs :

	Classe A	Classe B	Classe C	Classe D	Classe E
privateField	V				
packageField	V	V	V		
protectedField	V	V	V	V	
publicField	V	V	V	V	V

http://thecodersbreakfast.net

2. Construction et initialisation des objets dérivés

En Java, le constructeur de la classe dérivée doit prendre en charge l'intégralité de la construction de l'objet.

Le constructeur de la classe de base est la première instruction qui sera exécutée dans le constructeur de la classe dérivée.

2.1. La classe de base et dérivée possède au moins un constructeur

L'appel du constructeur de la classe dérivée s'effectue grâce à la méthode super().

```
public class A {  // constructeur
private int x ;
private int y ;
public A  (int p1, int p2)
{
x=p1 ;
y=p2 ;
}
……
}

public class B extends A {  // constructeur
private int z ;
public B  (int a, int b, int c)
{
super(a,b) // appel du constructeur de la classe dérivée
z=c ;
}
……
}
```

La situation donnée ci-dessus est la plus usuelle, dans laquelle la classe de base et la classe dérivée disposent toutes les deux d'au moins un constructeur public.

2.2. La classe de base ou super classe ne possède pas de constructeur

```
public class A {  // constructeur
protected int x ;
protected int y ;
…….……// aucun constructeur
}

public class B extends A {  // constructeur
private int z ;
public B  (int a, int b, int c)
{
super() ; // appel du constructeur par défaut par défaut de la classe dérivée
// x=a ;y=b ;
z=c ;
}
…….……
}
```

On appelle le constructeur par défaut dans la classe dérivée

2.3. La classe dérivée ne possède aucun constructeur

Lors de la création d'un objet du type dérivée celui-ci va lancer le constructeur par défaut de la classe de base. Cette situation peut vite mener à des problèmes non prévus, prenons les deux exemples suivants.

Soit la classe de base possède un constructeur par défaut

```
public class A {  // constructeur
private int x ;
private int y ;
public A  (int p1, int p2) {…….}
public A  () {…….}// constructeur par défaut

……..……..…..
}

public class B extends A {  // constructeur
……..……..// aucun constructeur
……..……..…..
}
B obj1 = new B () ;// Aucun problème le constructeur de A par défaut est lancé
```

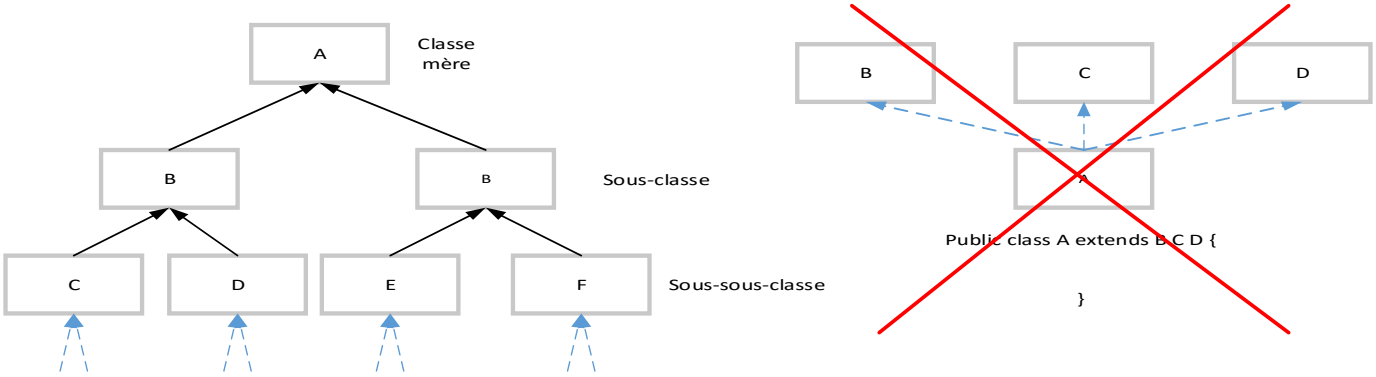
Soit la classe de base ne possède pas de constructeur par défaut :

```
public class A {  // constructeur
public A  (int p1, int p2) {…….}// pas de constructeur par défaut
……..……..…..
}
public class B extends A {  // constructeur
……..……..// aucun constructeur
……..……..…..
}
B obj1 = new B () ;// Problème le constructeur par défaut n'existe pas et au moins une méthode
constructeur dans la classe de base existe
```

On retiendra qu'une classe de base dans ce cas particulier doit :

- posséder au moins un constructeur public sans argument "constructeur par défaut", lequel sera alors appelé.
- posséder aucun constructeur ; il y aura appel du pseudo-constructeur par défaut.

La dérivation successive est possible en langage JAVA seul l'héritage multiple n'est pas autorisé :



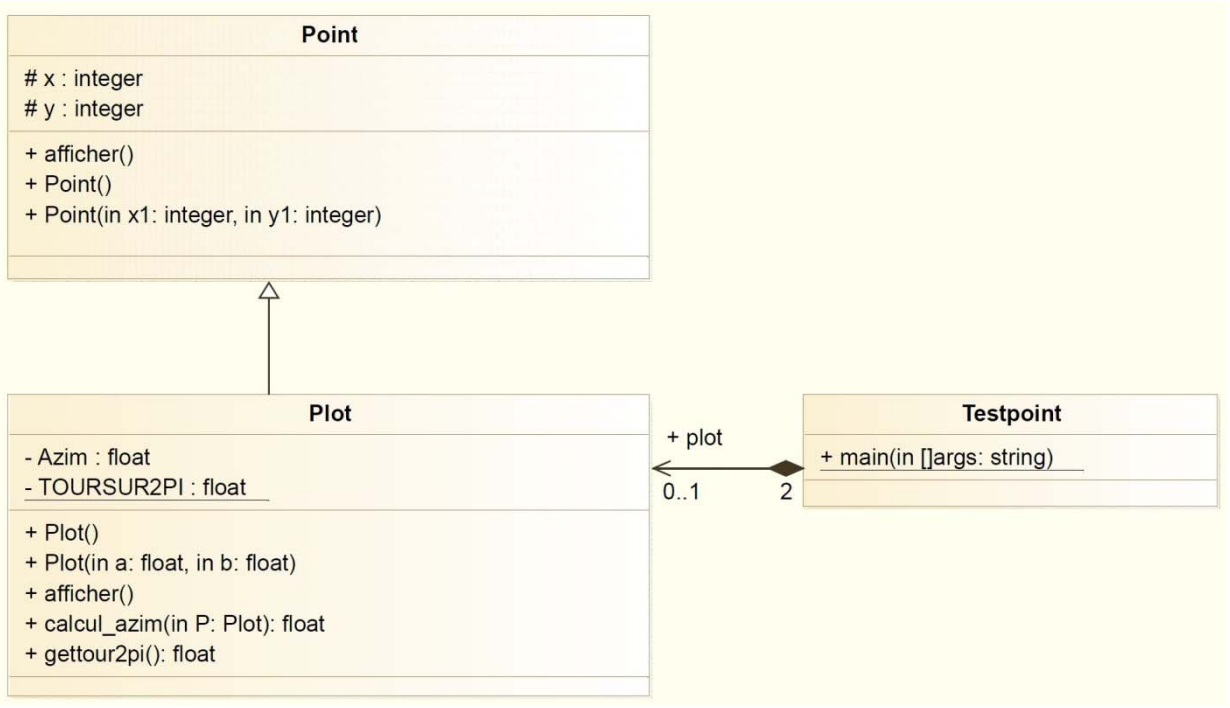
3. Redéfinition de méthodes

Une classe dérivée ou une classe de base peuvent posséder toutes les deux une méthode de même nom avec les mêmes paramètres d'entrées et de sorties. Par défaut, C'est la méthode de la classe dérivée qui sera appelée sur tout objet de la classe dérivée, masquant en quelque sorte la méthode de la classe de base. Pour qu'un objet de la classe dérivée puisse appeler la méthode commune de la classe de base il faut précéder l'appel du mot clé super.

```
public class A extends B {
……..…..
void fct1 () {……..……..……..…..}
float fct2 () {
        super.fct1() ;// appel de la méthode de la classe de base et non de la classe dérivée
        }
}
Public class B {
……..……..……..……..…
void fct1 () {……..……..……..……..…}
}
```

TD N°5 L'héritage en JAVA

On souhaite compléter la classe Point par une classe Plot qui hérite de la classe Point. Le diagramme de classe est le suivant :



Java utilise le mot clé super pour appeler une méthode de superclasse.

Ces applications développées sous Java utilisent des composants graphiques définis dans deux bibliothèques graphiques :

- AWT (Abstract Windowing Toolkit) première bibliothèque graphique développé sous JAVA
- SWING **Il a été développé durant l'existence de Java 1.1 et fait désormais partie des API centrales de Java**

Ces deux librairies fournissent des classes pour représenter des éléments d'interface comme les fenêtres, des boutons, des boîtes combo, des arborescences, des grilles et des menus - toute chose nécessaire à la construction d'une interface utilisateur dans une application Java. Pour cela, vous disposez du paquetage java.awt et javax.swing (ainsi que ses nombreux sous-paquetages).

Complétez la classe Plot et Point en adéquation avec le diagramme de classe. On fera attention lors de l'écriture de la classe Plot aux deux difficultés suivantes :

- L'appel du constructeur de la classe de base à l'intérieur du constructeur de la classe dérivée.
- La gestion de la redéfinition de la méthode affiche lors de l'écriture de la méthode affiche de la classe dérivée

Code source :

Classe Point

```
public class Point {
----------------------------int x;
-----------------------------int y;
public Point (){
x=0;
y=0;
}
public Point (int a, int b){
x=a;
y=b;
}
public void afficher()
{
System.out.println(" je suis un pt de coord " +x+ " "+y) ;
}
}
```

Classe testplot

```
public class testplot {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		float result;
		Plot P1= new Plot(1,2);
		Plot P2= new Plot(0,1);
		result=P1.calcul_azim(P2);
		System.out.println(" la valeur de l'azimuth est " +result) ;
		result=P1.calcul_azim(new Plot(1,1));
		System.out.println(" la valeur de l'azimuth est " +result) ;
		P1.afficher();
		P2.afficher();
	}

}
```

Classe Plot

```
import java.util.Scanner;
import  java.lang.Math.*;


public class Plot _________________________ {
	private float Azim;
	public static final float TOURSUR2PI=57.2957795131f;

	public void afficher () // afficher les coordonnées cartésiennes du plot
	{
		_____________________________________;

	}


public static float gettour2pi()
	{
		return TOURSUR2PI;
	}
```

```
	public  Plot() // intialiser les cordonnées à 0, 0
	{


		==========;

	}

	public  Plot(int a, int b)
	{
		==========;

	}

	public float calcul_azim(Point P)
	{  float result;

	if (P.x == 0)
		  if (P.y <= 0)
	    result = 0;
	   else
	    result = 180;

	   double enradian = Math.atan2(-((double)y-(double)P.y) ,((double)x-(double)P.x));

	   result   = (450 - ((float)enradian * TOURSUR2PI)) % 360;   //57.2957795

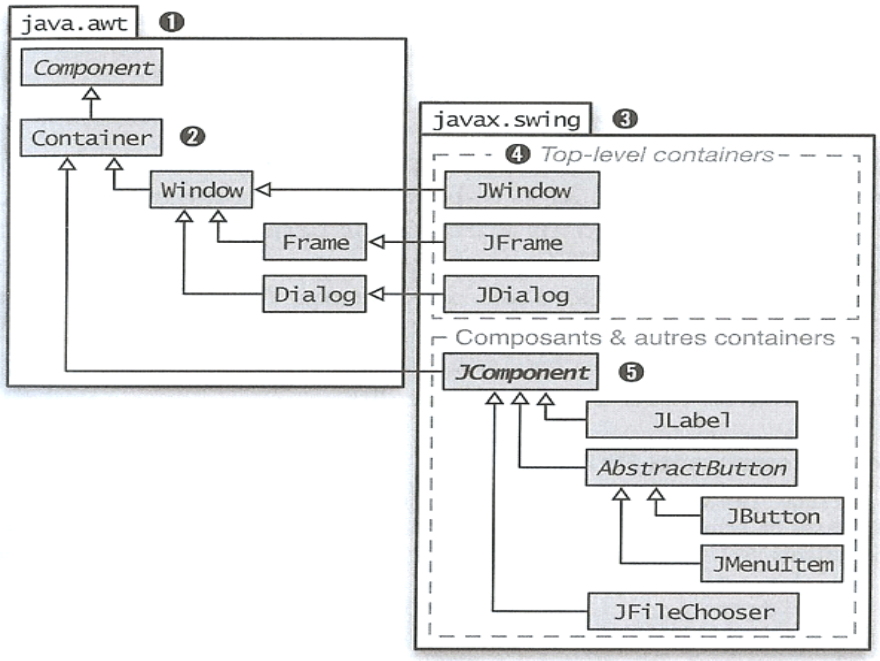

	return result;

	}
}
```

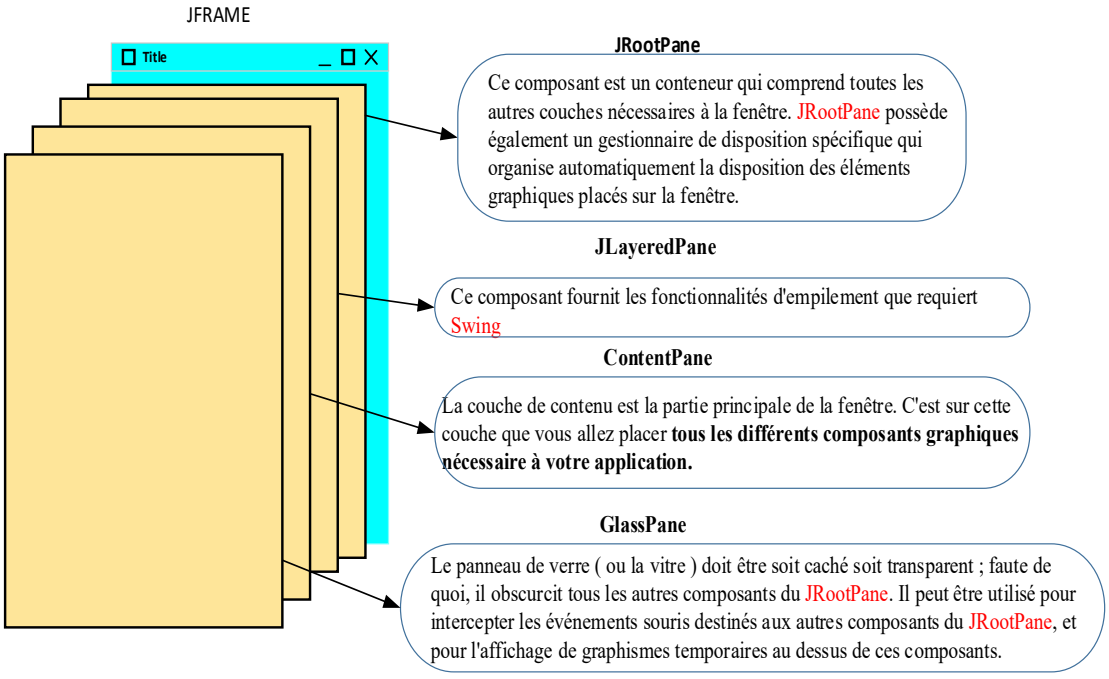
Initiation à la programmation graphique

Les objets graphiques (GUI) utilisent des composants graphiques définis dans la bibliothèque graphique **AWT** (*Abstract Windowing Toolkit*) ou **SWING**. La bibliothèque AWT ou SWING est un package du JDK (*Java Development Kit*), qui propose un ensemble d'outils de création d'applications graphiques. Nous utiliserons par la suite la bibliothèque SWING

La plupart des classes de composant Swing commencent par la lettre J : JButton, JFrame, etc. Ce sont des classes comme Button (bouton) et Frame (fenêtre) , mais il s'agit dans ce dernier cas des composants AWT. Pour retrouver ces composants Swing, vous devez importer le paquetage javax.swing.
En réalité JFrame hérite de Frame alors que les autres composants comme JButton, JLabel (étiquette), JTextField (champ de texte), JMenu (menu), etc. héritent tous de la classe de base JComponent qui fait partie de Swing, qui hérite lui-même indirectement de Component qui est un élément de AWT.



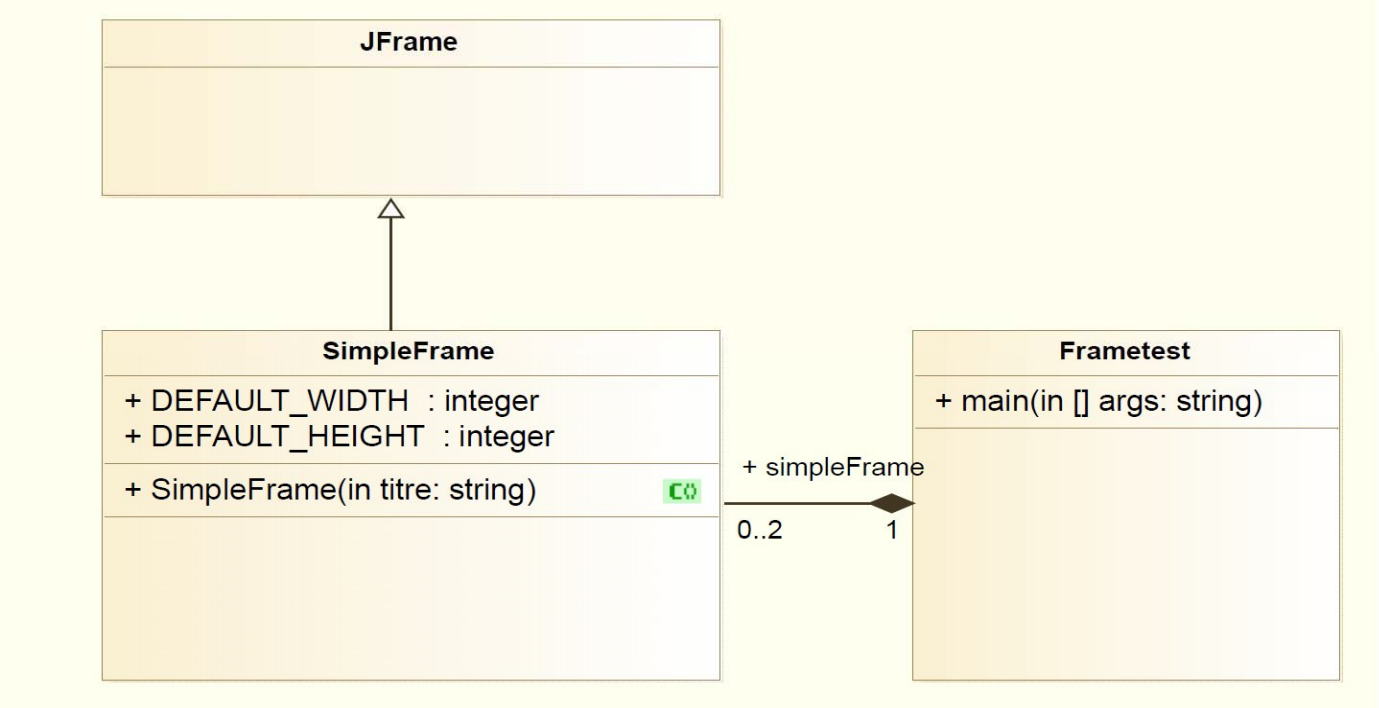
Une fenêtre en JAVA est un objet du type JFrame. Une fenêtre est constituée de quatre couches superposées comme on peut le constater en observant la figure ci-dessous



La plupart du temps, nous n'avons pas à nous préoccuper de la racine (JRootPane), de la couche supperposée (JLayeredPane) et de la vitre (GlassPane) ; elles sont nécessaires pour l'organisation de la barre de menus et du contenu. La partie qui intéresse les programmeurs Swing est plus généralement la couche contenue (**ContentPane)**. Pour récupérer l'objet représentant la couche ContentPane on utilisera la méthode **getContentPane()** qui nous renvoie l'objet du type ContentPane

TD N°6 : L'héritage pour la programmation graphique (Création d'une fenêtre et d'un panneau)

Soit les classes suivantes :



Classe SimpleFrame.java

```
import javax.swing.*;
import java.awt.*;
import java.util.Scanner;
import javax.swing.JFrame;



public class SimpleFrame extends JFrame {

  public SimpleFrame(String titre)
  {

  super(titre);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
  getContentPane().setBackground(Color.GREEN);

  }
  public static final int DEFAULT_WIDTH = 300;
  public static final int DEFAULT_HEIGHT = 200;
}
```

Classe Frametest.java

```
import javax.swing.*;

public class Frametest {

  public static void main(String[] args) {

  SimpleFrame frame = new SimpleFrame("fenetre");
  frame.setVisible(true);

  }
}
```

a) Identifiez l'héritage mise en place dans les classes ci-dessus

✍

b) Les méthodes setDefaultCloseOperation, setSize, getContentPane(),setVisible sont issues de quelle classe ?

✍

c) Lancez votre programme et observez

Soit la classe suivante :

```
import java.awt.Graphics;

import javax.swing.*;


public class Cercle extends JFrame {

  public  Cercle(){

  setTitle("fenetre");
  setSize(300,200);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  getContentPane().add(new Panneau());

  }

  public static void main(String[] args) {

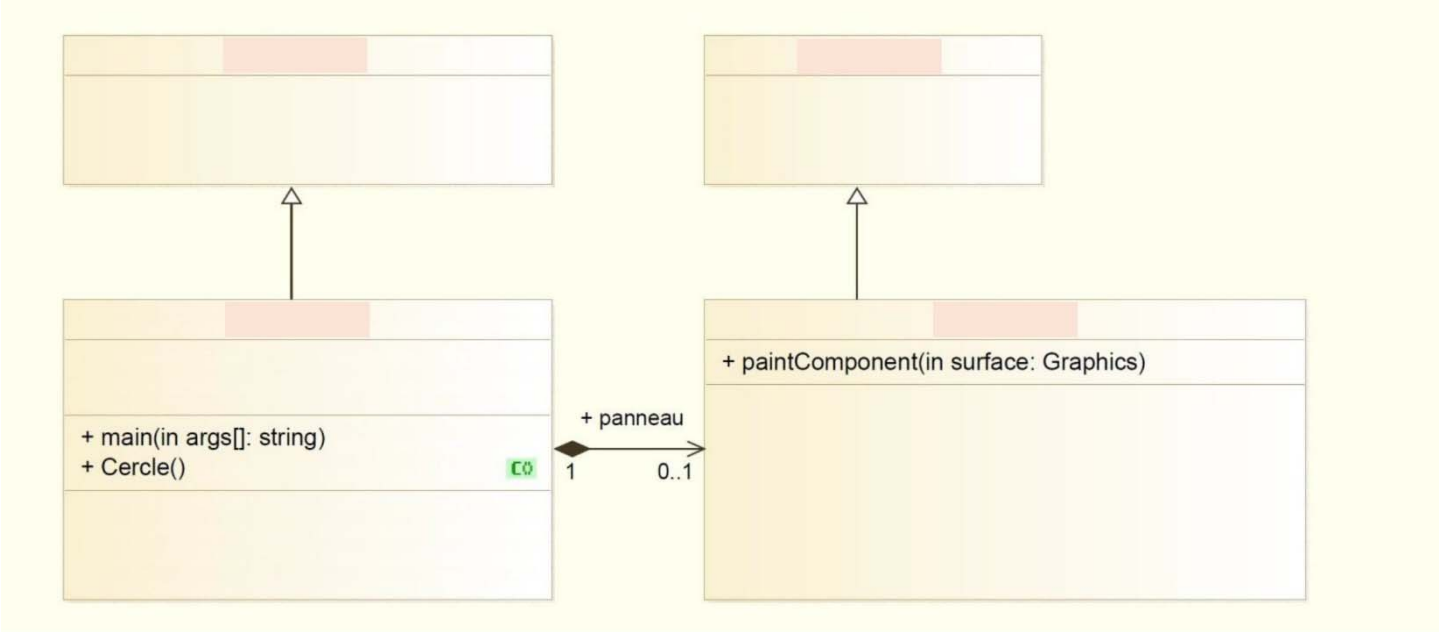
            new Cercle().setVisible(true);
            }

  class Panneau extends JPanel
  {
  public void paintComponent(Graphics surface){

            super.paintComponents(surface);

            surface.drawOval(50,50,200,100);
  }}}
```

A partir du code précédent, complétez le diagramme de classe suivant



a) Lancez et observez !

b) Quel est le constructeur utilisé pour la création de l'objet de la classe de base JFrame ?

✍

c) Pourquoi utiliser le mot clé super dans l'instruction super.paintComponents(surface) ?

✍

d) De quelle classe est issue la méthode drawOval ou paintComponent ?

✍

e) En vous servant du dossier ressource dessinez une chaine de caractère à l'intérieur de l'objet oval

On peut remarquer que la classe Panneau est défini à l'intérieur de la classe Cercle. C'est une particularité de JAVA la classe Panneau est appelé classe interne et obéit aux règles suivantes :

Une classe est dite interne lorsque sa définition est située à l'intérieur de la définition d'une autre classe. Deux raisons justifient l'emploi de classes internes :

- *Les méthodes des classes internes peuvent accéder aux attributs de la classe conteneur, à partir de l'envergure où elles sont définies, y compris les attributs qui pourrait être des données privées.*
- *Les classes internes peuvent être cachées aux autres classes du même paquetage, ce qui offre une meilleure protection sur des données sensibles.*

Ressources sur le dessin : la classe Graphics

Comment dessiner sur un JPanel (ou tout autre composants) ?

1. Solution 1 : redéfinission de la méthode paintComponent

Il y a deux techniques de dessin. La première donnée ici permet d'ajouter des graphismes sur des composants graphiques. Pour cela, on redéfinit la méthode paintComponent. Cette méthode est appelée chaque fois que l'objet doit être redessiné. Un objet est redessiné :

- par l'utilisateur via l'appel à la méthode repaint() (panel.repaint(); avec panel un objet de la classe Panneau1),
- automatiquement à chaque fois que l'objet graphique doit être redessiné (fenêtre redimensionnée, réduction-agrandissement, passage d'une autre fenêtre au-dessus de l'objet,…)

```
import java.awt.*;
import javax.swing.*;

public class Panneau1 extends JPanel {

String message= "test java";

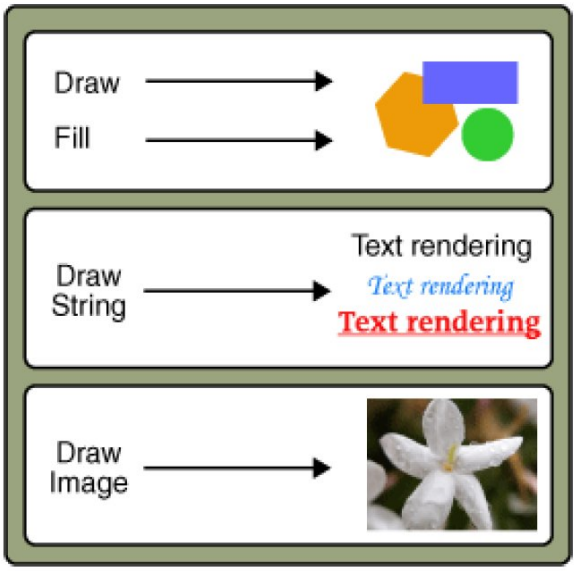
public void paintComponent(Graphics g)
{

super.paintComponent(g);
g.setColor(Color.BLUE);
g.setFont(new Font("SansSerif", Font.BOLD + Font.ITALIC, 24));
g.drawString(message,20,20);
g.setColor(Color.RED);
g.drawRect(0,0,10,10);

}
```

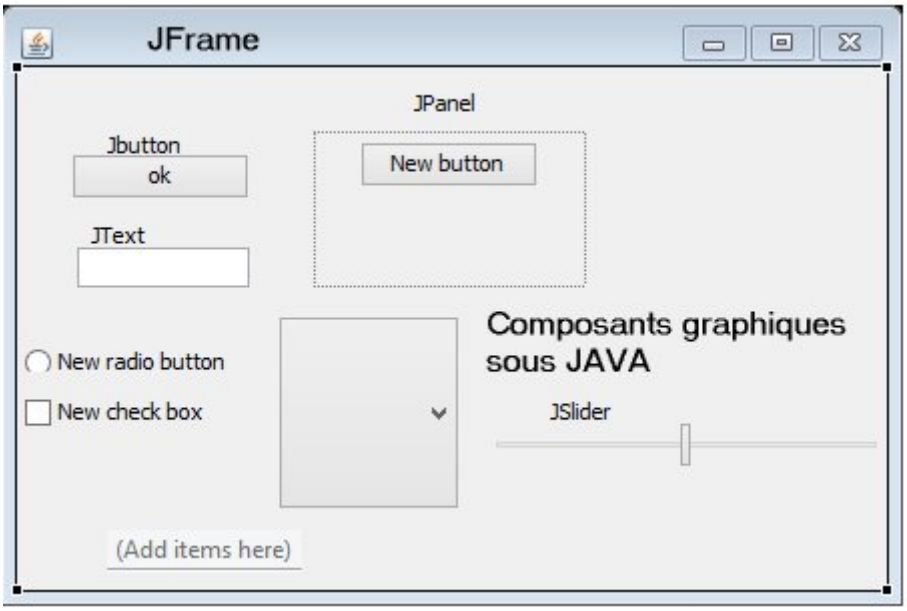
Dans le code ci-dessus, la classe Panneau1 hérite de la classe JPanel. Dans la classe Jpanel , la méthode paintComponent existe et est appelée par java à chaque fois que l'objet doit être affiché (redimensionnement de la fenêtre parente par exemple).
Dans notre classe Panneau1 nous redéfinissons la méthode paintComponent (surcharge de la méthode) pour que celle-ci affiche un carré rouge et une chaîne de caractère bleu. Le dessin se fait via l'appel aux méthodes de la classe Graphics qui est la classe permettant de dessiner sur n'importe quel objet graphique.
La classe java.awt.Graphics permet de dessiner des formes simples (lignes, rectangles, ovales), d'afficher du texte et des images.
Dans la fenêtre principale (Cadre1) il suffit ensuite de créer un objet panel de la classe Panneau1 et d'ajouter cet objet à notre fenêtre (méthode contentPane.add).

setColor() Fixe la couleur des futurs tracés
drawRect() Dessine un rectangle
drawOval() Dessine un ovale
drawLine() Dessine une ligne
drawString() Dessine une chaîne



TD N°7 : Les composants graphiques

Un composant est l'objet fondamental d'une IHM sous Java. Tout ce que nous voyons sur l'écran d'une application Java est un composant. Ceci comprend des fenêtres, des boutons, des cases à cocher, des barres de défilement, des listes, des menus, des champs de saisie, etc.



Pour être utilisé, un composant doit généralement être placé dans un conteneur.

Un conteneur de composant est un objet pouvant accueillir d'autres composants graphiques.
Les trois types de conteneurs les plus utilisés en JAVA sont : JFrame, JPanel et JApplet.

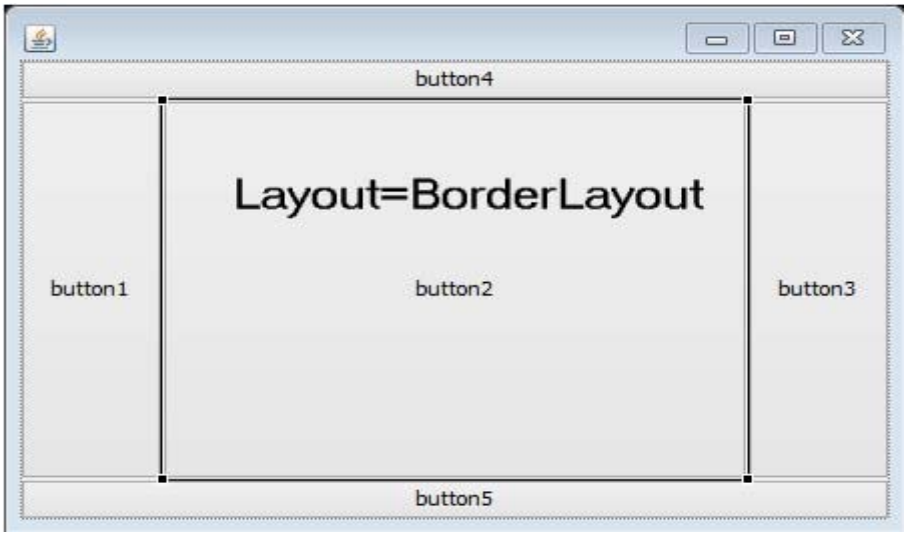
Un conteneur possède un layout. Un Layout, est un type de placement automatique des objets graphiques.

On retrouve plusieurs types de Layout :

Java.awt.FlowLayout	Dispose les composants d'un Container les uns derrière les autres en ligne et à leur taille préférée, en retournant à la ligne si le container n'est pas assez large
Java.awt.GridLayout	Dispose les composants d'un container dans une grille dont toutes les cellules ont les mêmes dimensions
Java.awt.BorderLayout	Dispose cinq composants maximum dans un container, deux aux bords supérieur et inférieur à leur hauteur préférée et un au centre qui occupe le reste de l'espace
Java.awt.CardLayout	Affiche un composant à la fois parmi l'ensemble des composants d'un container
Java.swing.BoxLayout	Dispose les composants en ligne à leur hauteur préférée ou en colonne à leur largeur préférée
Java.awt.GridBagLayout	Dispose les composants d'un container dans une grille dont les cellules peuvent avoir des dimensions variables

Le BorderLayout permet un placement maximum de 5 composants selon les 4 positions cardinales plus le centre.

Swing utilise des gestionnaires de placement (appelé aussi gestionnaire de disposition) pour disposer des composants à l'intérieur de conteneurs en contrôlant leur taille et leur position. Les gestionnaires de placement définissent une stratégie de disposition des composants au lieu de spécifier des positions absolues.



L'exemple suivant permet de mettre en évidence le concept de layout en utilisant le `BorderLayout`

```
import java.awt.Graphics;
import java.awt.*;
import java.awt.event.*;
import java.text.DateFormat;
import java.util.Date;
import javax.swing.*;

public class Layout_ex extends JFrame {

	public Layout_ex(){
	setTitle("fenetre");
	setSize(300,200);
	setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	//getContentPane().add(new Panneau());
	getContentPane().setLayout(new BorderLayout(0, 0));
	JButton btnNewButton_1 = new JButton("button4");
	getContentPane().add(btnNewButton_1, BorderLayout.NORTH);

	JButton btnNewButton_2 = new JButton("button1");
	getContentPane().add(btnNewButton_2, BorderLayout.WEST);

	JButton btnNewButton_3 = new JButton("button2");
	getContentPane().add(btnNewButton_3, BorderLayout.CENTER);

	JButton btnNewButton_4 = new JButton("button3");
	getContentPane().add(btnNewButton_4, BorderLayout.EAST);

	JButton btnNewButton_5 = new JButton("button5");
	getContentPane().add(btnNewButton_5, BorderLayout.SOUTH);
	}

	public static void main(String[] args) {
		new Layout_ex().setVisible(true);


	}
```

A partir de cet exemple on souhaite intégrer un objet panneau dans la fenêtre (voir page N° 47 méthode add). Ce panneau qui est un conteneur intégrera lui aussi un objet du type BoardLayout (méthode setlayout) ainsi que les cinq boutons précédents (méthode add).

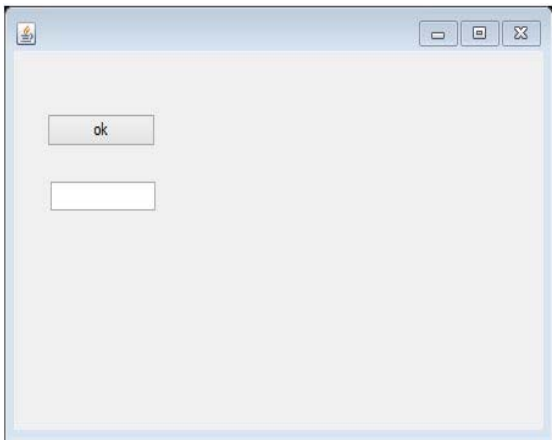
Modifiez et complétez le programme précédent afin qu'il réponde à l'application envisagée.

```
import java.awt.*;
import java.awt.event.*;
import java.text.DateFormat;
import java.util.Date;
import javax.swing.*;
public class Layout extends JFrame {
public Layout(){
setTitle("fenetre");
setSize(300,200);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JPanel panel1----------------------------------; // création d'un panneau
getContentPane()_____________________________;
-------------------------------------------------
```

```
-------------------------------------------------
}
public static void main(String[] args) {
new Layout().setVisible(true);
// TODO Auto-generated method stub
}
class Panneau extends JPanel
{
public void paintComponent(Graphics surface){
}
}
}
```

On remarquera que l'élaboration d'une IHM nourrie de nombreux composants graphiques peut devenir très complexe. Afin de nous aider dans l'élaboration de cet IHM, le logiciel Eclipse intègre un logiciel de développement d'application rapide (windows builder).

En vous servant du tutorial de prise en main du logiciel Eclipse, réalisez l'interface graphique suivante via windows builder:



En cliquant sur le bouton ok, aucun événement ne se produit. En effet nous n'avons pas géré le gestionnaire d'événement et la programmation événementielle. La programmation événementielle fait appel à la notion d'interface en JAVA. Avant d'aborder cette nouvelle notion, nous allons nous intéresser à la notion de classe abstraite.

```
package swing;

import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import javax.swing.JTextField;

public class Testswing_ex extends JFrame {

	private JPanel contentPane;
	private JTextField textField;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					Testswing_ex frame = new Testswing_ex();
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}
```

```
	/**
	 * Create the frame.
	 */
	public Testswing_ex() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 450, 300);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);

		JButton btnOk = new JButton("ok");
		btnOk.setBounds(65, 74, 89, 23);
		contentPane.add(btnOk);

		textField = new JTextField();
		textField.setBounds(68, 132, 110, 23);
		contentPane.add(textField);
		textField.setColumns(10);
	}
}
```

La programmation événementielle en JAVA : Notion d'interfaces et de classes abstraites

1. Les classes et les méthodes du type final

Le mot clé final pour un attribut interdit la modification de cet attribut (const en C++). Ce concept peut s'élargir aux méthodes ou aux classes :

```
public final void fct1() {
...
}
```

Une méthode déclarée final ne peut pas être redéfinie dans une classe dérivée.

Une classe peut aussi être déclarée final :

```
public final class A {

//Méthodes et attributs

}
```

Dans ce cas la classe ne pourra plus être dérivée. Certaines classes de la JDK sont déclarés comme final (exemple de la classe Math.

2. Les classes abstraites

A mesure que l'on remonte dans la hiérarchie des classes, celles-ci deviennent plus générales et souvent plus abstraites. A un certain moment, la classe ancêtre devient tellement générale qu'on la considère surtout comme un moule pour des classes dérivées et non plus comme une véritable classe dotée d'instances.

Une classe abstraite est une classe qui ne permet pas d'instancier ou de créer des objets. Elle ne servira que de classe de base à une classe dérivée. La syntaxe est la suivante :

```
abstract class A {


//Méthodes et attributs


}

A obj1 ;  // la déclaration d'une classe abstraite est possible

Obj1= new A() ; Impossible on peut ne peut pas créer un objet à partir d'une classe abstraite
```

Classe abstraite et héritage

```
public class B extends A {

//Méthodes et attributs apportant une fonctionnalité supplémentaire à la classe A

}

B obj2= new B() ;  // création de l'objet obj2
A obj2= new B() ;  // Cette syntaxe reste possible car la classe B est une sous classe de la classe A
```

On peut aussi trouver des méthodes abstraites dans une classe, c'est-à-dire des méthodes qui ne sont pas définies dans la classe.

```
abstract class A {

public void fct1(){ String S ; …… ;if(…)…. ;} // La méthodes est définie dans la classe
public abstract double fct2( byte e) ;// La méthode n'est pas définie dans la classe, elle est abstraite

}
```

Dès qu'une classe abstraite comporte une ou plusieurs méthodes abstraites, elle est abstraite, et ce même si l'on n'indique pas le mot clé abstract devant sa déclaration (ce qui reste quand même vivement conseillé).
Ceci est correct :

```
public class A {  // Cette classe est abstraite

public void fct1(){ String S ; …… ;if(…)…. ;}
public abstract double fct2( byte e) ;//

}
```

Une classe dérivée d'une classe abstraite n'est pas obligée de (re)définir toutes les méthodes abstraites de sa classe de base (elle peut même n'en redéfinir aucune).

On peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- ✓ soit sous forme d'une implémentation complète de méthodes (non abstraites) et de champs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
- ✓ soit sous forme de méthodes abstraites dont on est alors sûr qu'elles existeront dans toute classe dérivée instanciable.

Les sous classes d'une classe abstraite restent abstraites tant que la sous classe ne redéfinit pas toutes les méthodes abstraites de la super classe abstraite.

3. Les interfaces

Comme son nom l'indique, l'interface joue le rôle d'intermédiaire entre l'application et l'objet sollicité.
Une interface correspond donc à une classe qui définit non pas un modèle d'objet (une sorte de moule) mais un ensemble de comportements possibles, sans que ces comportements soient réellement décrits.
Plus précisément, lorsqu'un utilisateur clique par exemple sur le bouton « Valider » d'une application, une action doit être menée. Cette action diffère selon l'application. Il peut s'agir de confirmer l'envoi d'un message électronique ou encore de supprimer un fichier. La classe modélisant le bouton de validation doit donc proposer une méthode nommée par exemple actionArealiser() (en anglais actionPerformed()) sans décrire explicitement le code de cette action. Seul le programmeur est à même de décrire l'action à réaliser.
L'interface permettra de lier des méthodes ou des constantes communes à des classes non liées par une relation d'héritage

Déclaration d'une interface

La déclaration d'une interface se présente comme celle d'une classe. On utilise simplement le mot clé interface à la place de class :

```
Interface intf {
// constantes
void methode1 ( int d) ;// déclaration des prototypes de méthodes
void methode2 ( int e) ;// déclaration des prototypes de méthodes
int volume_max=3 ;// déclaration d'une constante
}
```

Les méthodes d'une interface sont :

- *abstraites (puisque nous ne fournissons pas de définition)*
- *publiques (puisqu'elles devront être redéfinies plus tard).*

Les constantes d'une interface sont forcément :

- public
- static
- final

Un objet du type interface ne peut pas être créé, on ne peut pas instancier une interface directement.

~~Intf i = new intf() ;~~

Implémentation d'une interface

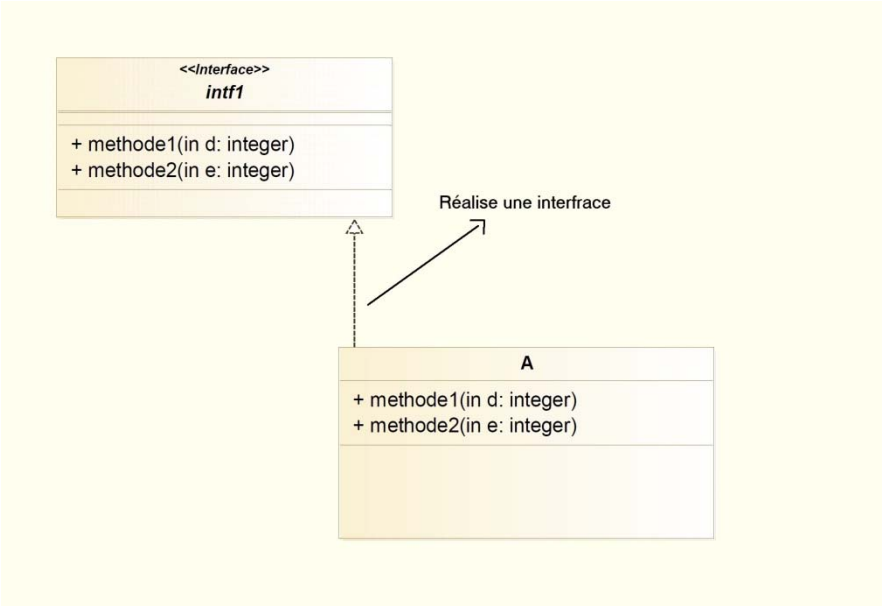
Une fois l'interface définie, les méthodes sont concrètement décrites au sein des différentes classes qui implémentent l'interface. Pour cela, vous devez :

- ✓ placer le mot clé implements lors de la création de la classe ;
- ✓ décrire ce que réalisent toutes les méthodes de l'interface (définition des méthodes).
- ✓ La définition des méthodes peut être différente d'une classe à une autre

public class A **implements** intf {

public type methode1(int d) {
// Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
}
public type methode2(int e) {
// Ici sont décrites les actions à mener pour cette méthode au sein de cette classe
}
// + autres méthodes et attributs de la classe A

}



A obja = new A() ;

Intf obja = new A() ; //car on considère qu'une interface attribue un type supplémentaire à une classe d'objet

Utilisation d'une interface

Si une classe A implémente une interface Intf, une sous-classe B de A implémente aussi automatiquement Intf ; une instance B pourra être référencée par une variable de type Intf et bénéficiera, au moins par héritage, des définitions des méthodes prototypées dans Intf.

public class B **extends** A {

void methodedeB (string k) { ……methode1(4) ;……}

}

Intf objB = new B() ;

Une même classe peut implémenter plusieurs interfaces :

public class A **implements** intf, intf1 {

}

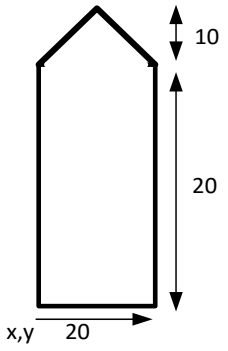
Dans ce cas d'utilisation, les interfaces permettent de compenser, en grande partie, l'absence d'héritage multiple.

TD N°8 : Les interfaces

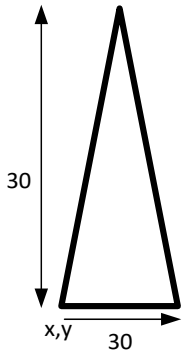
L'application suivante va permettre de créer le dessin de plusieurs objets sur un panneau :

- Objet transpondeur A
- Objet transpondeur B
- Texte

Ces objets et les symboles graphiques associés sont les suivants :



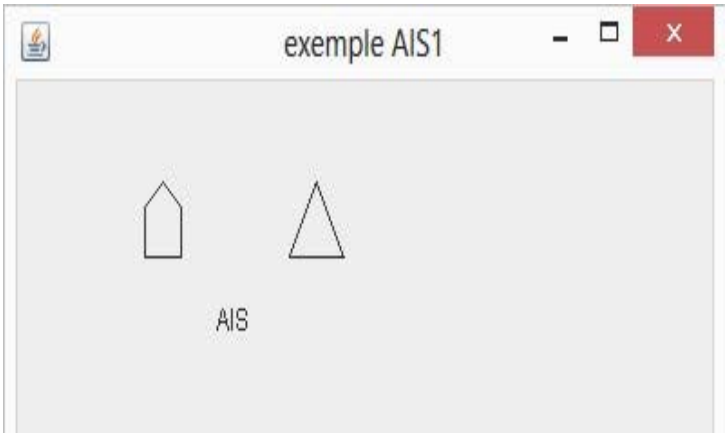
Transpondeur
AIS
Type classe B



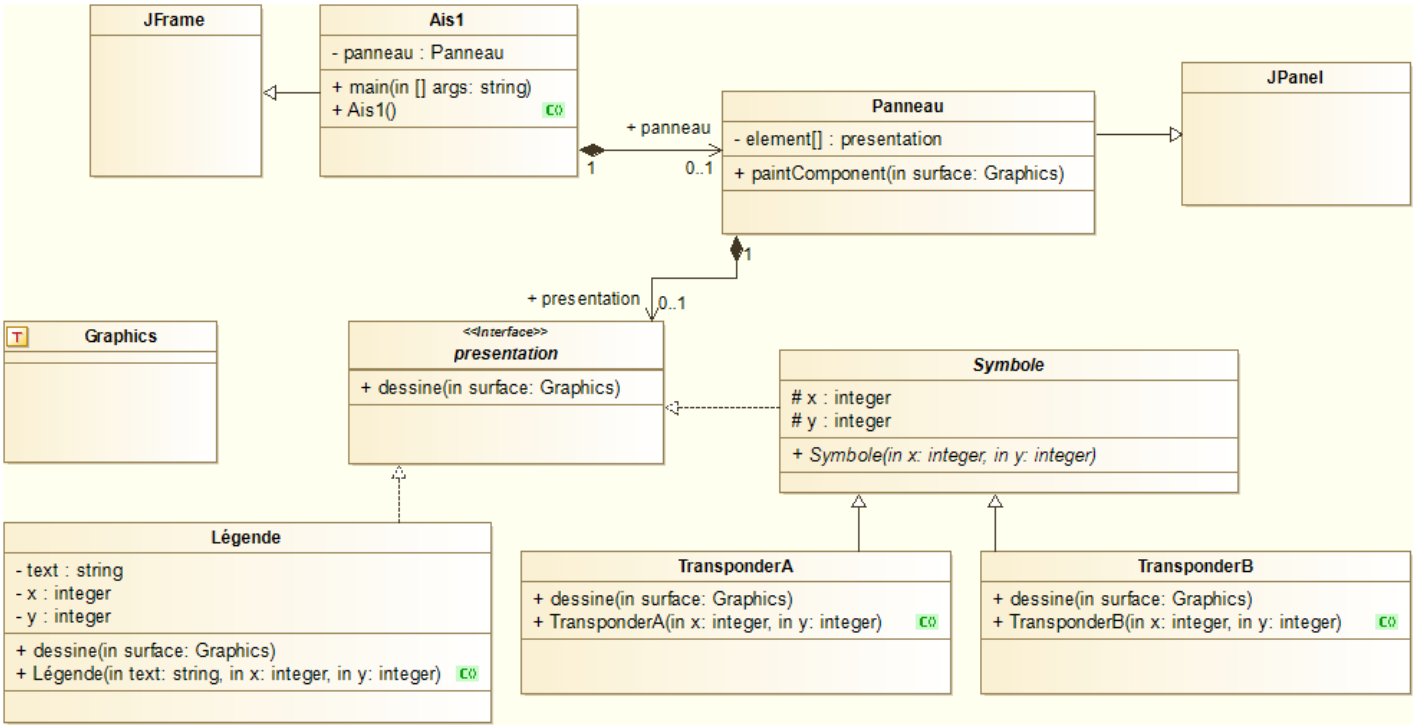
Transpondeur
AIS
Type classe A

AIS : Automatic identification system

Une fois notre applicatif développé en Java celui-ci fournira à l'exécution la fenêtre suivante :



Le diagramme de classe est le suivant :



L'interface presentation permet de mettre en connexion plusieurs hiérarchies de classes (Légende, Symbole ….), par l'intermédiaire de méthodes spécifiques déterminées par l'interface, comme ici la méthode dessine.

L'interface presentation permet de compenser, en grande partie, l'absence d'héritage multiple en JAVA.

Les classes TransponderA et TransponderB héritent de la classe Symbole qui dispose de la méthode dessine. Par ailleurs, la classe Légende qui est totalement indépendante de la première hiérarchie dispose également de la méthode dessine. Pour être sûr de dessiner un objet quelconque sur une fenêtre, par exemple, il suffit de passer par l'interface Présentation qui dispose de la méthode Dessine et qui spécifie donc que les objets associés à cette interface vont obligatoirement implémenter cette méthode.

La classe Panneau disposera d'un tableau nommé element du type presentation constitué de trois cases :

private presentation element [3] ;

Chaque case pourra accueillir un objet du type légende, TransponderA, transponderB.

1) Complétez la définition de la méthode dessine des classes TransponderA, transponderB, Légende.

drawPolygon(int[] lesX, int[] lesY, int nombrePoint) Dessine un polygone et le ferme en joignant les extrémités
fillPolygon(int[] lesX, int[] lesY, int nombrePoint) Dessine un polygone plein
drawString(String chaine,int x,int y); // dessine une chaine à partir de la position x et y

2) Créez l'interface présentation
3) Créez dans la classe Panneau un membre ou un attribut tableau de trois éléments :
   - portée private
   - nom du tableau element
   - type des éléments presentation
   - création et initialisation du premier élément du tableau un objet du type TransponderA de coordonnées x=70 y=70 ;
   - création et initialisation du deuxième élément du tableau un objet du type TransponderB de coordonnées x=150 y=70 ;
   - création et initialisation du troisième élément du tableau un objet Légende de coordonnées x=110 y=110 ;Texte= "AIS"

On rappelle que dans le cas simple d'un tableau d'entier la création et l'initialisation du tableau s'effectue de la manière suivante :

int [] tab={1,3,2} ; Initialisation et création d'un tableau de trois cases.

4) Créez dans la méthode main un objet obj1 du type Ais1. L'objet Ais1 sera visible (appel de la méthode .setVisible(**true**)).
5) Testez votre programme

Code source :

```
import java.awt.Graphics;
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;


public class Ais1 extends JFrame  {

      private JPanel panneau = new Panneau();
             public Ais1() {
                super("exemple AIS1");
                //setTitle("titre de la fenêtre");
                setSize(400, 400);
                //addWindowListener(new WindowAdapter() {
                //   public void windowClosing(WindowEvent evenement) {
                //     System.exit(0);  }
                getContentPane().add(panneau);
                panneau.setLayout(null);



}

             public static void main(String[] args) {
                  _____________________________
                  _____________________________


                }
}
```

```
            class Panneau extends JPanel{
                    public void paintComponent(Graphics surface){
                   super.paintComponent(surface);
                   for (int i=0; i<3; i++) element[i].dessine(surface);
                   //surface.drawOval(10, 80, 120, 120);

                        }
                // Déclaration et initialisation du tableau
                    private final _____________________
                            ___________________________,
                            ___________________________,
                            ___________________________
                    };
                }

           // Création de l'interface

 interface ____________________________________


abstract class Symbole implements presentation {
      protected final int x,y;
      public Symbole(int x, int y){
            this.x=x;
            this.y=y;
      }
      }


class TransponderA extends Symbole{
      public      TransponderA (int x, int y){super(x,y); }
      public void dessine (Graphics surface){
            ____________________________________________________________________
      }

      }


      class TransponderB extends Symbole{
            public      TransponderB (int x, int y){super(x,y); }
            public void dessine (Graphics surface){
                  ____________________________________________________________
            }
                        }
```

```
class Légende implements presentation{
            public      Légende (String chaine, int x, int y){
                  this.chaine=chaine;
                  this.x=x;
                  this.y=y;
                   }
            public void dessine (Graphics surface){
                  surface.drawString(chaine, x, y);
            }
            private final String chaine;
            private final int x,y;
            }
```

4. Les interfaces et la programmation événementielle

4.1. Présentation

La gestion des événements est d'une importance capitale pour les programmes ayant une interface utilisateur graphique constitués de multiples composants graphiques.

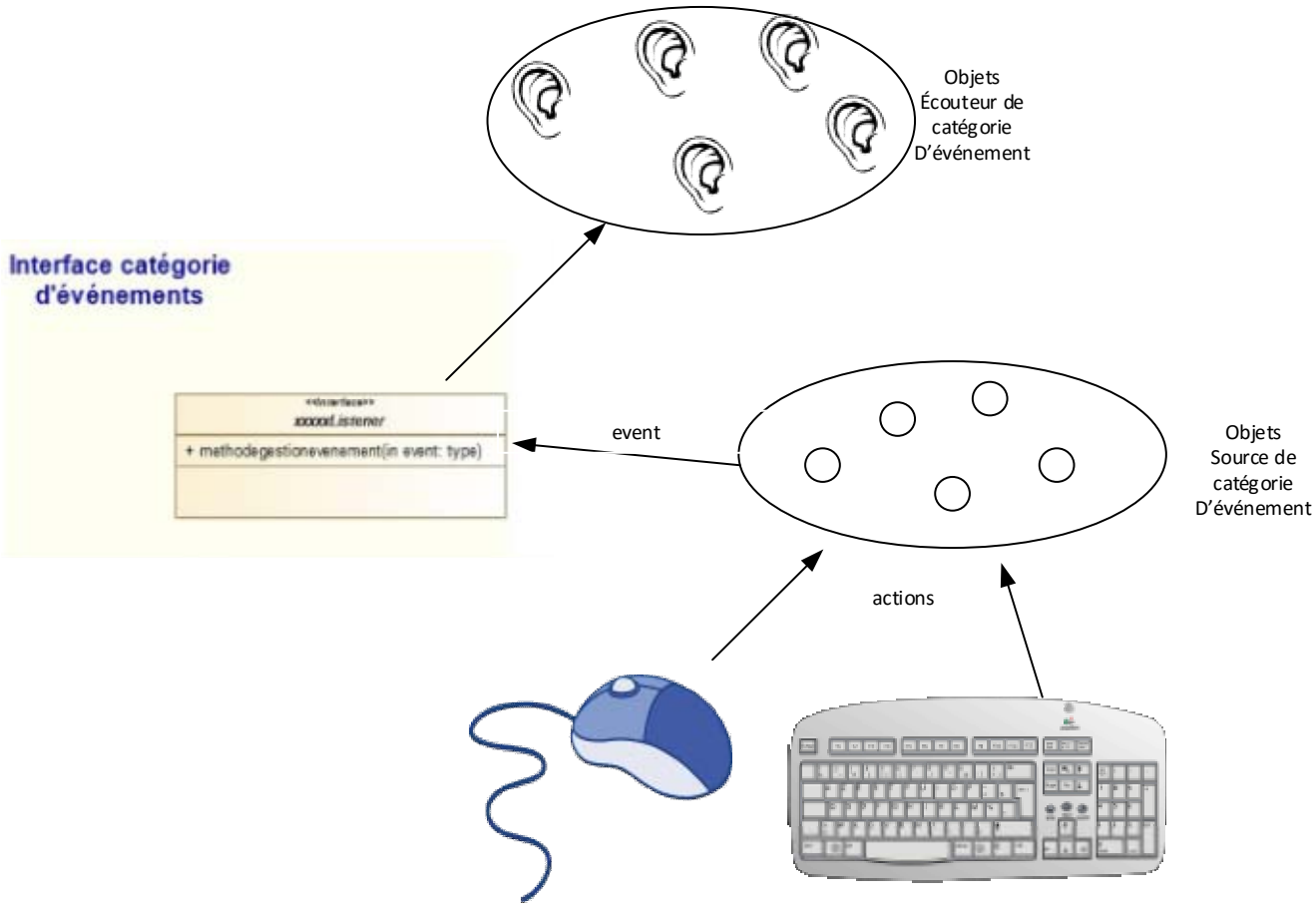
Chaque composant graphique (bouton, onglet, boite de texte ...) peut être source d'événement vis à vis d'une action effectué par exemple sur la souris ou le clavier.

On doit disposer d'un objet qui soit capable de capturer et de gérer ces événements (un écouteur d'événement).

La programmation événementielle est basée sur le modèle de communication objets source d'événement et objet écouteur d'événement au travers d'une interface xxxxListener

Lorsqu'un événement se produit, l'objet source d'événement envoie l'objet événement (event) à tous les objets écouteurs recensés via l'interface.

Les objets écouteurs utilisent alors l'information contenue dans l'objet événement (event) pour déterminer l'action à réaliser que l'on code dans la méthode redéfinie de l'interface.



Dans ce modèle de communication quelques règles s'imposent :

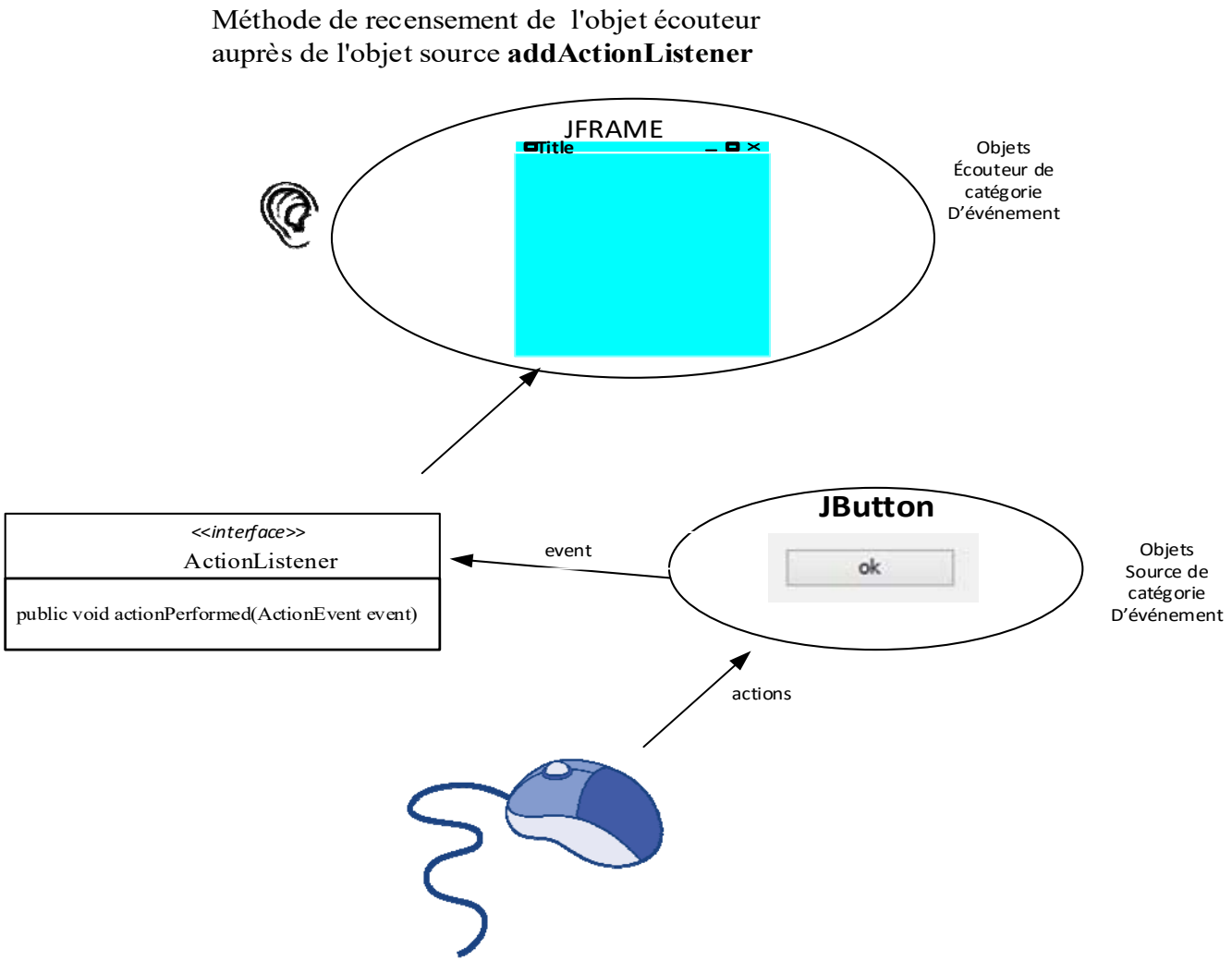
1) Un objet écouteur est une instance d'une classe qui **doit implémenter une interface spéciale appelée interface écouteur (listener interface)**. Le choix de cette interface dépend de la catégorie d'événements à gérer. L'objet écouteur va permettre de capturer et de gérer les événements en utilisant la ou les méthodes de l'interface.
2) Pour l'objet écouteur, l'implémentation d'une interface signifie qu'il faut redéfinir les méthodes prévues dans l'interface (respectant ainsi le contrat) avec la signature correcte de ces méthodes.
3) Pour recenser l'objet écouteur auprès de l'objet source, on utilise l'instruction suivante :

ObjetSourceEvénement**.add**Evénement**Listener(**objetEcouteurEvénement**).**

4.2. L'interface ActionListener

Appliquant ces quelques règles à une catégorie d'événement lié à l'interface ActionListener qui permet de détecter le click d'une souris.

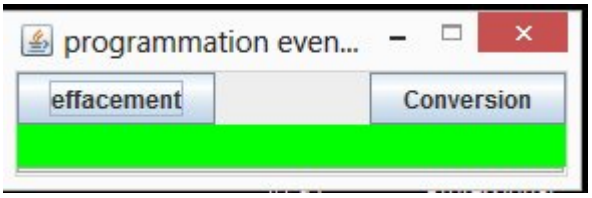
Le schéma précédent devient le suivant :



TD N°9 :Application: Interface ActionListener

On donne le programme ci-dessous d'une interface graphique très simple, deux objet du type bouton (Jbutton) et un objet champ de texte (JTextfield). On souhaite programmer un événement lorsqu'on click gauche sur le bouton conversion qui se traduira par l'affichage de la chaine de caractère " bonjour" dans l'objet champ de texte. On souhaite programmer un deuxième événement lorsqu'on click gauche sur le bouton effacement qui se traduira par l'affichage de la chaine de caractère "" dans l'objet champ de texte.

L'IHM sera sera la suivante :



On suivra les différentes étapes vues en 5.4.1) pour la mise en place de ce cas de programmation événementielle :

1) Choisir comme objet écouteur, l'objet fenêtre. L'interface sera du type ActionListener

2) Redéfinir la méthode actionPerformed afin de fournir les actions nécessaires pour chaque source d'événement. Pour déterminer qui a été l'objet source d'événement on utilisera la méthode getSource():

```
public void actionPerformed(ActionEvent e) {
        if ((e.getSource())==nomdel'objetsourced'evenement)
```

Pour afficher un texte dans un objet du type label champ de texte (classe JLabel) on utilisera la méthode suivante :

```
void setText(String chaine) ;
```

3) recenser les objets écouteurs auprès de l'objet source en utilisant la méthode addActionListener

Testez votre programme sous eclipse

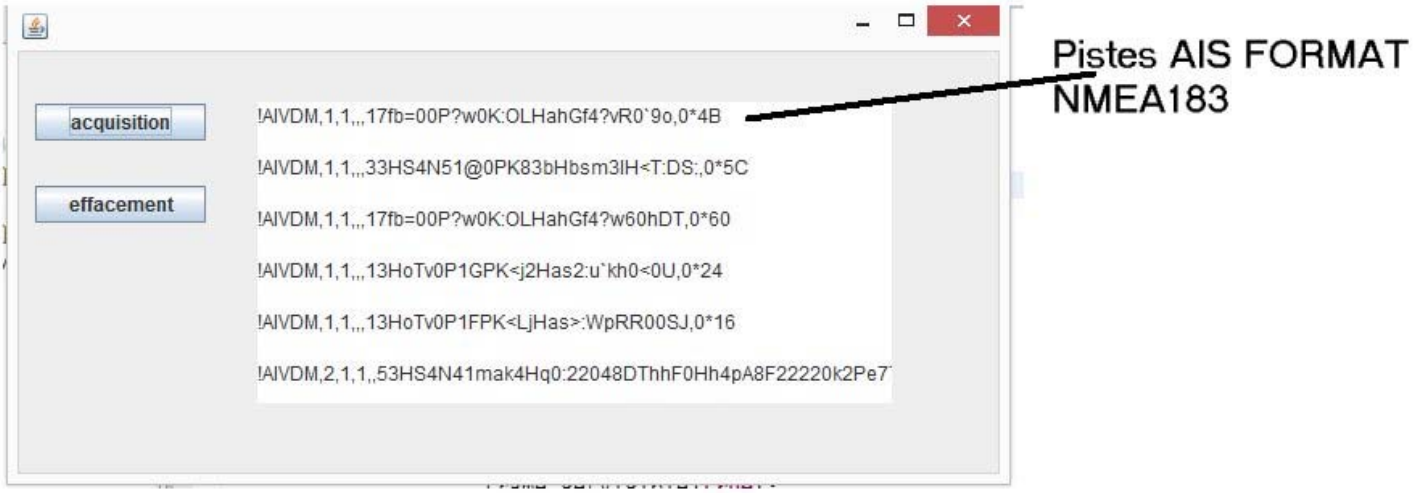
Code source :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class fenetre extends JFrame ----------------------------------{
private JButton conversion = new JButton("Conversion");
private JButton effacement = new JButton("effacement");
private JLabel résultat = new JLabel("");
private JPanel panneau = new JPanel();// Aucune necessite de créer une classe panneau
public fenetre() {
setTitle("programmation evenementielle");
setBounds(100, 100, 280, 80);
setDefaultCloseOperation(EXIT_ON_CLOSE);
panneau.setLayout(new BorderLayout());
panneau.add(conversion, BorderLayout.EAST);
panneau.add(effacement, BorderLayout.WEST);
add(panneau, BorderLayout.NORTH);
résultat.setHorizontalAlignment(JLabel.RIGHT);
résultat.setBorder(BorderFactory.createEtchedBorder());
add(résultat, BorderLayout.SOUTH);
getContentPane().setBackground(Color.GREEN);
setResizable(false);
setVisible(true);
```

```
}
```

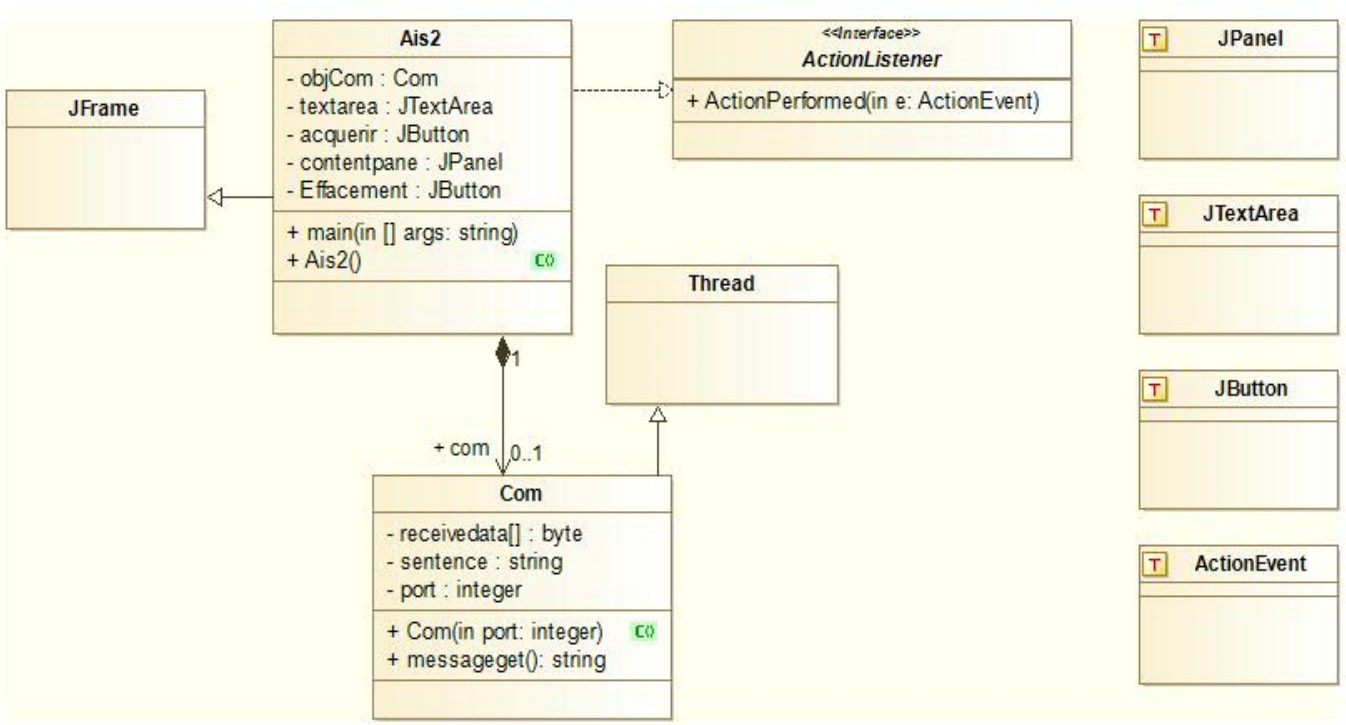
TD N°10 : Synthèse étude d'un cas : L'AIS

On souhaite à partir des connaissances abordées précédemment, créer une interface homme machine nous permettant d'exploiter les pistes issues d'un récepteur AIS. L'IHM sera constitué d'une zone de texte nous permettant d'afficher les pistes AIS au format NMEA183. Un bouton acquisition nous permettra de récupérer la piste AIS. Un bouton effacer permettra d'effacer le contenu du texte de la zone de texte. L'IHM ressemblera à la figure ci-dessous :



Le diagramme de classe sera constitué de deux classes principales, la classe Ais2 et la classe Com.

1) Réalisez sous windows builder la classe Ais2 en plaçant les objets graphiques



Le langage java est un langage modulaire qui permet d'intégrer facilement une classe déjà développée (classe Com). Une instance de la classe Com nous permettra grâce à un processus léger (thread) de récupérer en mode multitâche les pistes AIS qui circulent sur le réseau IP (datagramme UDP, port 10110).

2) Créez un nouveau fichier source java Com.java dont le contenu sera le suivant :

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


import java.io.IOException;
import java.net.*;

public class Com extends Thread{

//public static void main(String[] args) {
//      int port = args.length == 0 ? 57 : Integer.parseInt(args[0]);
//      new Receiver().run(port);
//   }
          private byte[] receiveData ;
          private String sentence;
          private int port;

          Com(int p ){
                    port=p;
          }

          public String getmessage(){

                    return sentence;
          }

   public void run() {
    try {
     DatagramSocket serverSocket = new DatagramSocket(port);
     byte[] receiveData = new byte[64];

     System.out.println("Listening on udp:" + InetAddress.getLocalHost().getHostAddress() + Integer.toString(port));
     DatagramPacket receivePacket = new DatagramPacket(receiveData,receiveData.length);

     while(true)
     {
          serverSocket.receive(receivePacket);
          sentence = new String( receivePacket.getData(), 0,receivePacket.getLength() );
          System.out.println("RECEIVED: " + sentence);
          // now send acknowledgement packet back to sender
          //InetAddress IPAddress = receivePacket.getAddress();
          //String sendString = "polo";
          //byte[] sendData = sendString.getBytes("UTF-8");
          //DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,IPAddress,receivePacket.getPort());
          //serverSocket.send(sendPacket);
     }
    } catch (IOException e) {
          System.out.println(e);
    }
    // should close serverSocket in finally block
   }
}
```

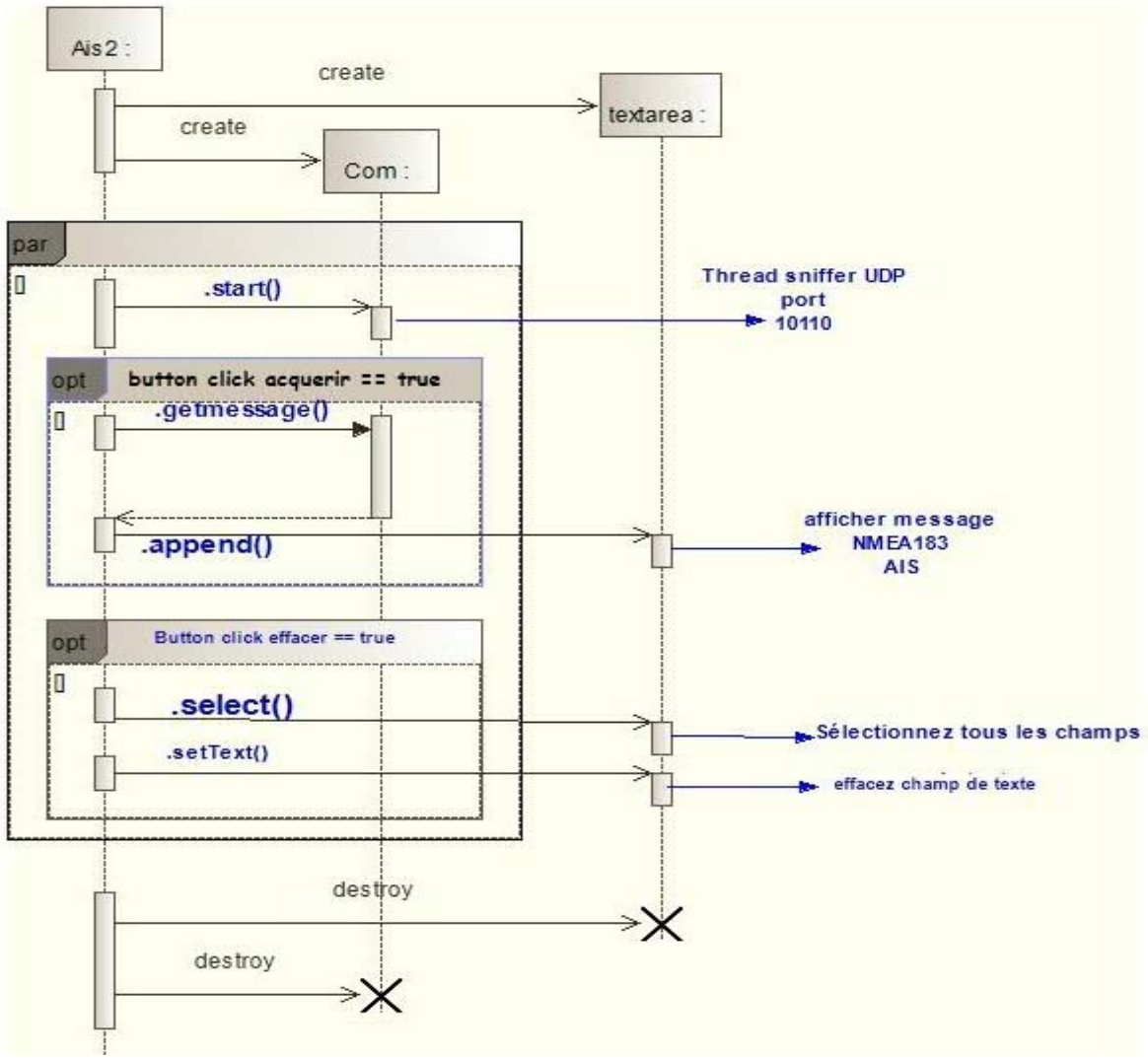
3) Décrire le rôle des deux méthodes :

- ✓ `Com(int p ) ;`
- ✓ `public String getmessage() ;`

4) Dans la classe Ais2.java et sa méthode constructeur, créez un objet nommé objCom du type Com avec comme port d'écoute le port 10110. L'instruction qui suivra la création de l'objet nous permettra de lancer le thread en utilisant la méthode start().
5) Lors de l'appui sur le bouton acquérir (événement onclick) on doit récupérer grâce à la méthode messageget() de la classe Com, la chaine de caractère qui correspond à une trame AIS. Cette chaine de caractère sera affichée dans la zone de texte de l'objet textarea. Donnez les lignes de code qui permettent de gérer cet événement (voir diagramme de séquence ci-dessous).

   Pour afficher une chaine de caractère dans un objet de la classe JTextArea :

void append (String message) // affichage sur une ligne d'une chaine de caractère classe JTextArea



6) Le bouton effacer doit permettre d'effacer le contenu de la zone champ de texte. Donnez les lignes de code qui permettent de gérer cet événement.

Code source :

```
void selectAll(void);// permet de sélectionner toute la zone de texte classe JTextArea
void setText(null); permet d'effacer toute la zone de texte sélectionnée classe JTextArea

import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import javax.swing.JTextArea;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class AIS2 extends JFrame {

	private JPanel contentPane;
	private String messageais;
	private Com objCom;
	private JTextArea textArea;
	private JButton btnNewButton_1;
	private JButton btnNewButton;
	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					AIS2 frame = new AIS2();
					frame.setVisible(true);
				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public AIS2() {
// creation du thread
```

```
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 527, 336);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
```

```
            setContentPane(contentPane);
            contentPane.setLayout(null);

             btnNewButton = new JButton("Acquisition");
            btnNewButton.addActionListener(new ActionListener() {
                 public void actionPerformed(ActionEvent arg0) {
```

```
                 }
            });
            btnNewButton.setBounds(21, 59, 89, 23);
            contentPane.add(btnNewButton);

             btnNewButton_1 = new JButton("effacement");
            btnNewButton_1.addActionListener(new ActionListener() {
                 public void actionPerformed(ActionEvent arg0) {
```

```
                 }
            });
            btnNewButton_1.setBounds(21, 120, 89, 23);
            contentPane.add(btnNewButton_1);

            textArea = new JTextArea();
            textArea.setBounds(135, 11, 355, 275);
            contentPane.add(textArea);
      }
}
```

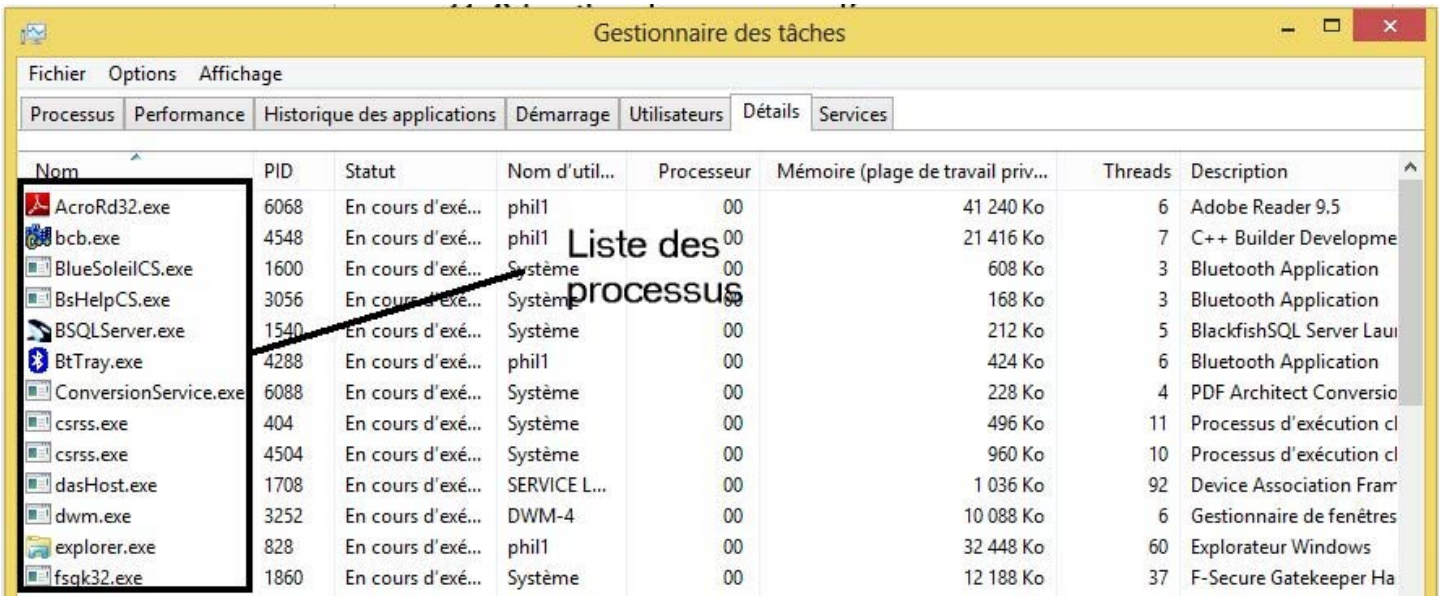
Les threads ou processus léger

1. Rappel les processus lourds

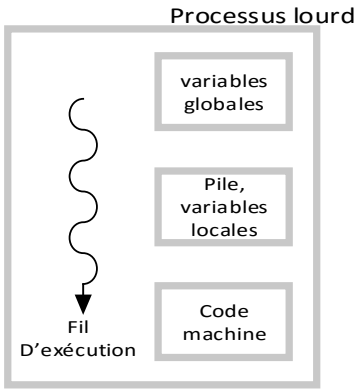
Un processus lourd est un programme en cours d'exécution. Chaque processus lourd implique la gestion d'un espace d'adressage virtuel MMU et de nouvelles copies de toutes les variables et ressources nécessaires à l'exécution (pile, registres, fichiers ouverts, verrous etc…).
Le principal avantage de ces processus est la protection mémoire entre les processus du même système.
Par contre, l'utilisation des processus lourds présente les inconvénients suivants :

- leur création nécessite des appels systèmes coûteux en temps,
- le changement de contexte entre processus est une opération lente, en particulier pour de nombreux transferts en mémoire,
- le coût des mécanismes de protection associés au processus,
- l'interaction, la synchronisation ou la communication entre processus nécessite l'utilisation de mécanismes de communication spéciaux (tube communicant appelé "*pipe*", *socket*, boîte aux lettres),
- le partage de mémoire entre processus s'effectue par ajout de mécanismes lourds (bibliothèque de partage de mémoire).

La liste des processus lourd est donnée par le gestionnaire des taches :

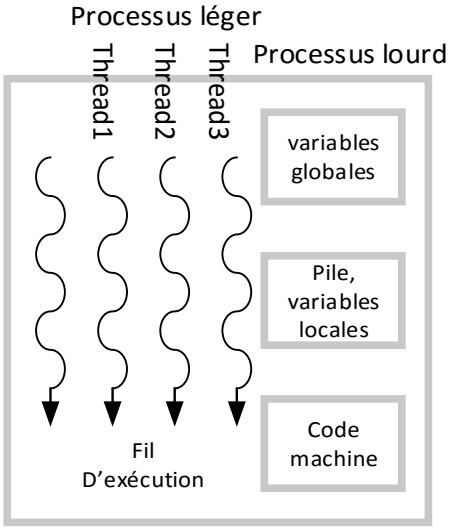


Un processus lourd classique, ne contenant qu'un seul fil d'exécution, est dit **monoprogrammé** :
L'exécution du code du processus est réalisée de manière séquentielle par un fil de contrôle (*thread of control*).

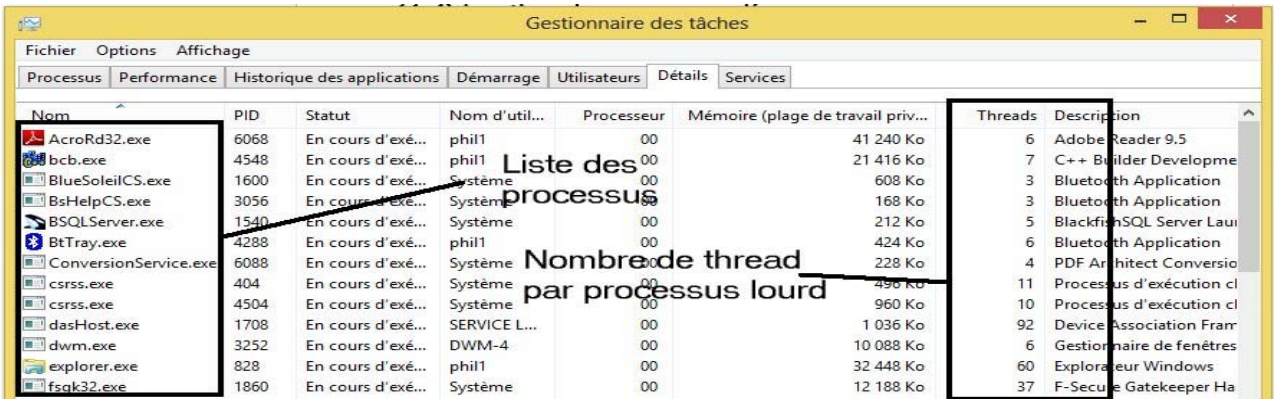


2. ***Les processus légers***

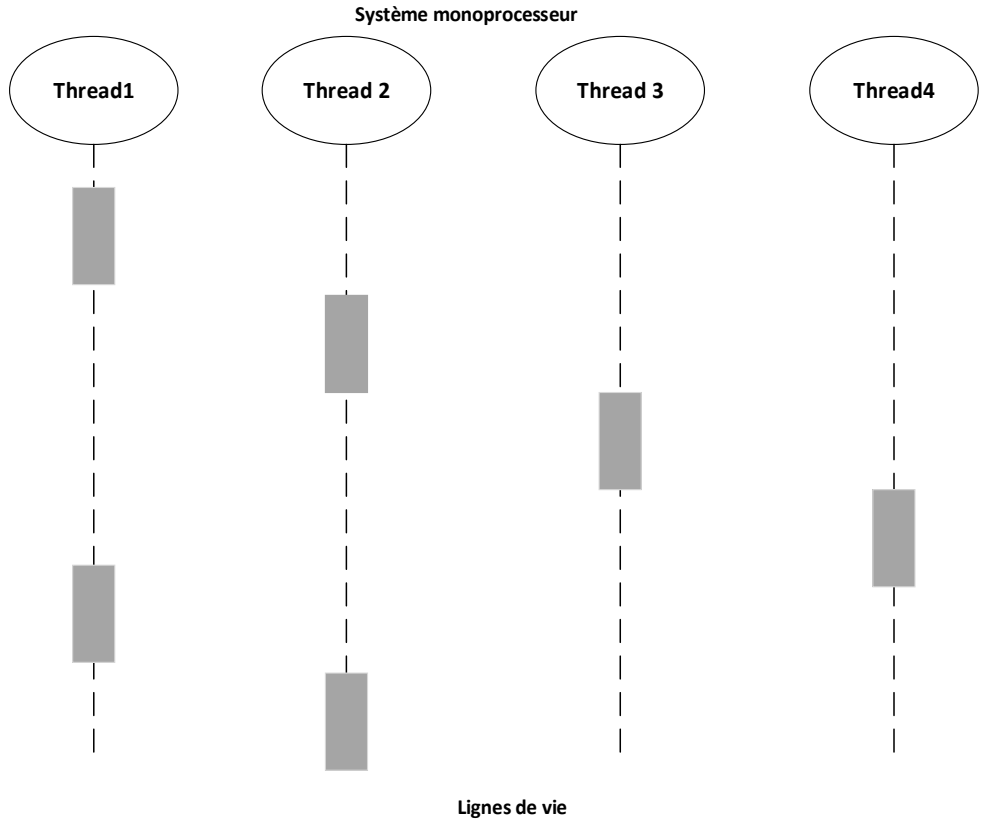
Les threads ou processus léger permettent de dérouler plusieurs blocs d'instructions, en PARALLELE, à l'intérieur du même processus. Un thread exécute une fonction ou méthode. Chaque fonction associée à un thread va être exécutée de façon parallèle et indépendante.



Le système d'exploitation gère donc des centaines de thread :



Dans un système monoprocesseur l'illusion sera donnée que chaque thread est exécuté de façon parallèle ou concurrente. En réalité chaque thread sera exécuté à tour de rôle mais très rapidement.



Un programme comporte toujours un thread dit thread principal correspondant tout simplement à la méthode main()

Avantages :

- **Multi-tâche moins coûteux** : puisqu'il n'y a pas de changement de mémoire virtuelle, la commutation de contexte (*context switch*) entre deux threads est moins coûteuse que la commutation de contexte de processus lourd

- **Communication entre threads plus rapide et plus efficace** : grâce au partage de certaines ressources entre threads, IPC (*Inter Processus Communication*) inutile pour les threads

Inconvénients :

Programmation utilisant des threads est toutefois plus difficile : obligation de mettre en place des **mécanismes de synchronisation**.

3. La nécessité d'un thread

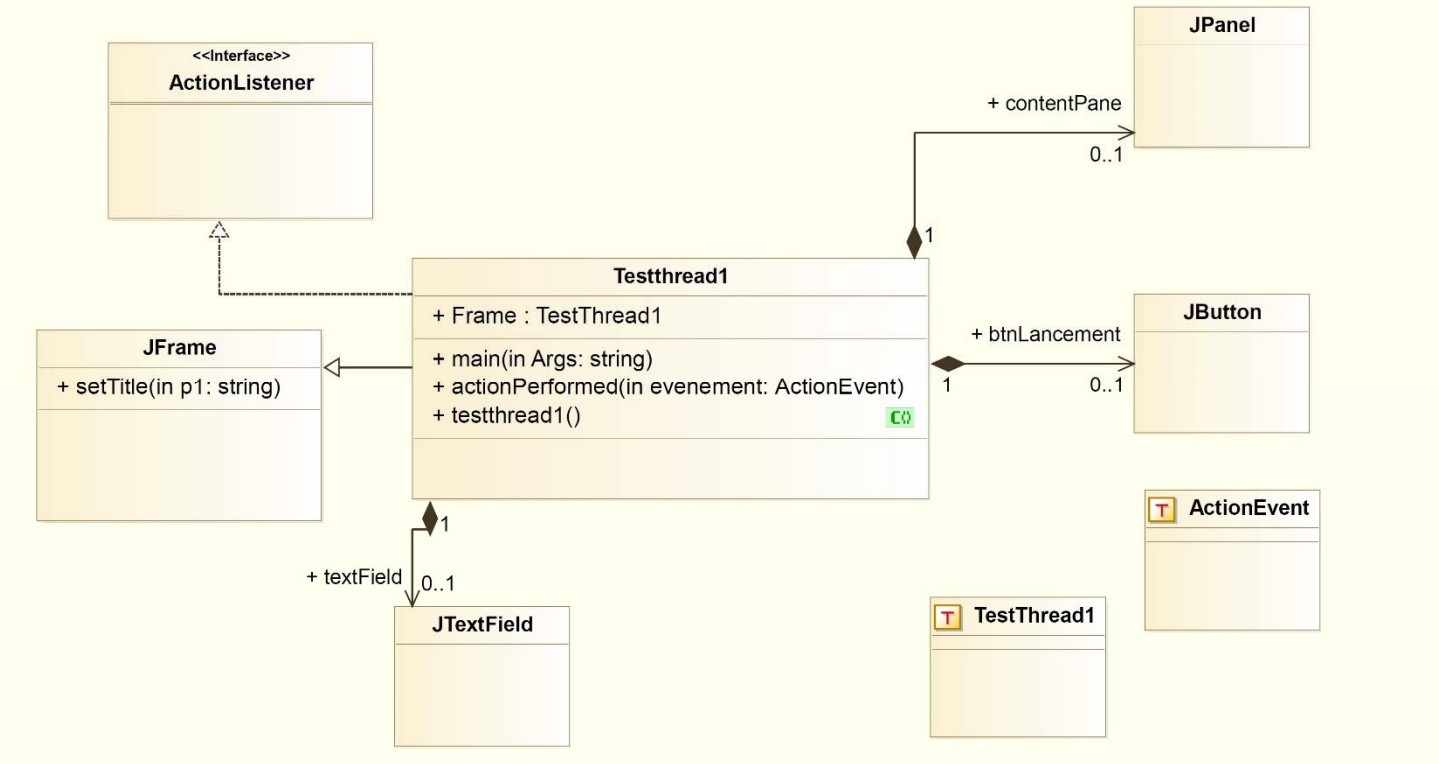
On souhaite développer un programme dont l'interface homme machine est donnée ci-dessous :



Un bouton permettra de lancer un compteur i (boucle for) de 0 à 100000. La valeur du compteur sera affichée dans le titre de la fenêtre. La méthode setTitle de la classe JFrame permettra d'afficher la valeur du compteur i dans le titre de la fenêtre.

```
this.setTitle("i="+i);
```

Une boite de texte permettra de saisir une chaine de caractère. Le diagramme de classe est donné ci-dessous :



Réalisez sous windowsbuilder cette IHM.

Qu'observez-vous lorsque le compteur s'incrémente et que vous essayez de compléter le champ de texte.

✍

Conclusion

Dans votre programme certains calculs ou traitements peuvent être long. Pour toutes les taches plutôt longues (interface graphique, web….) on utilisera des threads séparés

Code source :

```
import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import javax.swing.JTextField;
import java.awt.event.*;

public class testthread1 extends JFrame implements ActionListener  {

	private JPanel contentPane;
	private final JButton btnLancement = new JButton("lancement");
	private JTextField textField;

	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					testthread1 frame = new testthread1();
					frame.setVisible(true);

				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public testthread1() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 248, 173);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);
		btnLancement.setBounds(55, 11, 120, 31);
```

```
            contentPane.add(btnLancement);
            btnLancement.addActionListener(this);
            textField = new JTextField();
            textField.setBounds(55, 68, 120, 31);
            contentPane.add(textField);
            textField.setColumns(10);
      }
      public void actionPerformed(ActionEvent evenement) {

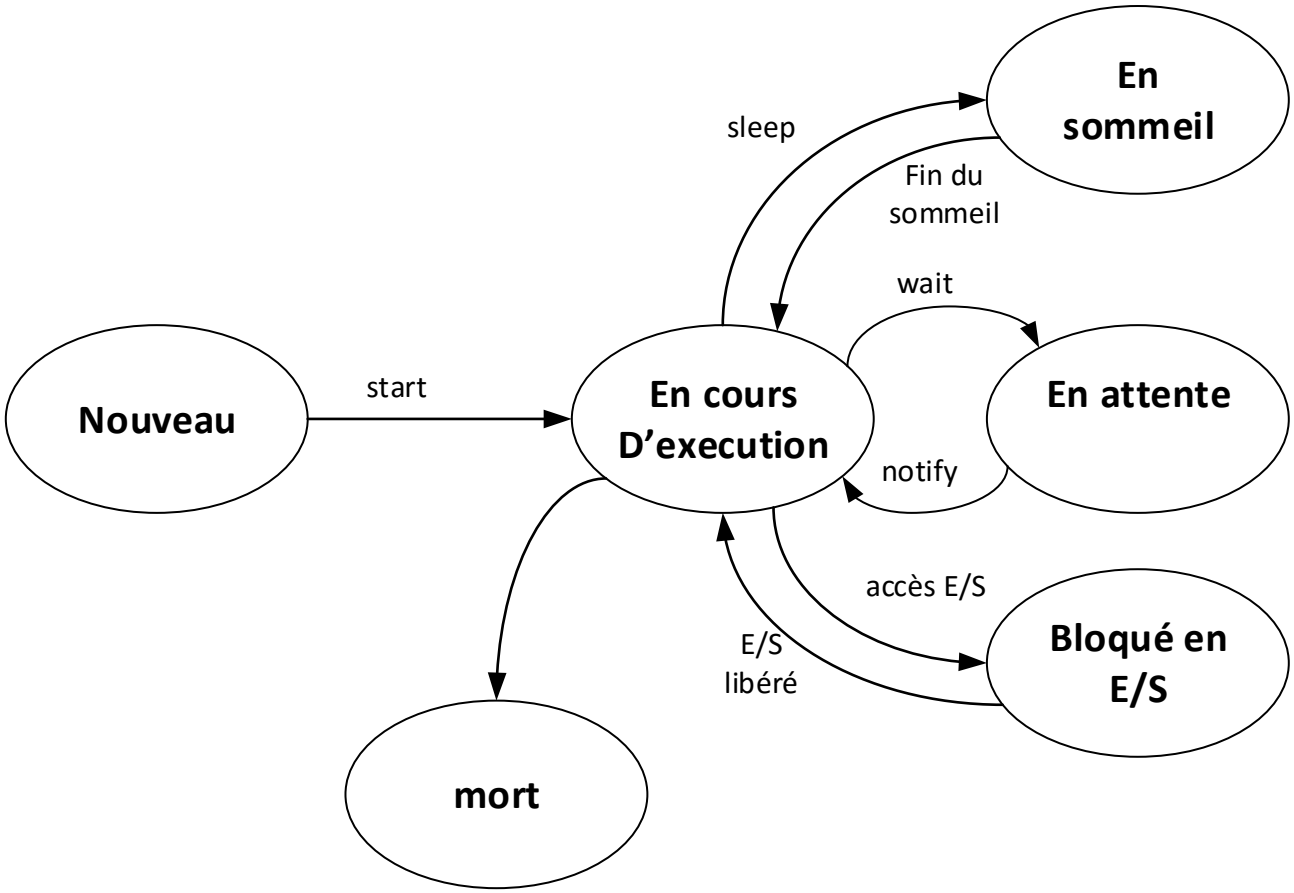
            for (long i=0; i<100000; i++)
                  this.setTitle("i="+i);

      }

}
```

4. Les principaux états d'un thread

Un thread peut être dans différents états comme résumé dans le diagramme ci-dessous :



Lorsqu'on créé un thread avec l'opérateur new, le thread n'est pas encore exécuté (**état nouveau**)
Une fois que l'on a invoqué la méthode start le thread prend l'état **en cours d'exécution**, il est en attente d'une fenêtre d'exécution fournie par le système d'exploitation.
Lorsqu'un thread est bloqué ou en attente il est temporairement inactif et n'exécute pas de code. Plusieurs types d'inactivités sont possibles **sommeil, attente, bloqué.**

Un thread peut se terminer (**état mort**) pour l'une des deux raisons suivantes :

- Il meurt d'une mort naturelle parce que la méthode run s'est normalement terminée
- Il meurt parce qu'une exception a été provoqué et a mis à la fin à la méthode run

5. Définition création et démarrage d'un thread

Lorsque vous créez un thread avec l'opérateur new le thread n'est pas encore exécuté. Cela signifie qu'il se trouve dans l'état Nouveau. Lorsqu'un thread se trouve dans cet état, le programme n'a pas encore commencé à exécuter le code qui se trouve dans le thread. Un thread est un objet d'une classe **qui dispose d'une méthode run qui sera exécutée lorsque le thread sera démarré.**

Pour exécuter notre programme dans un thread séparé, nous devons dériver la classe A de la classe Thread et définir **une méthode run** dans la classe

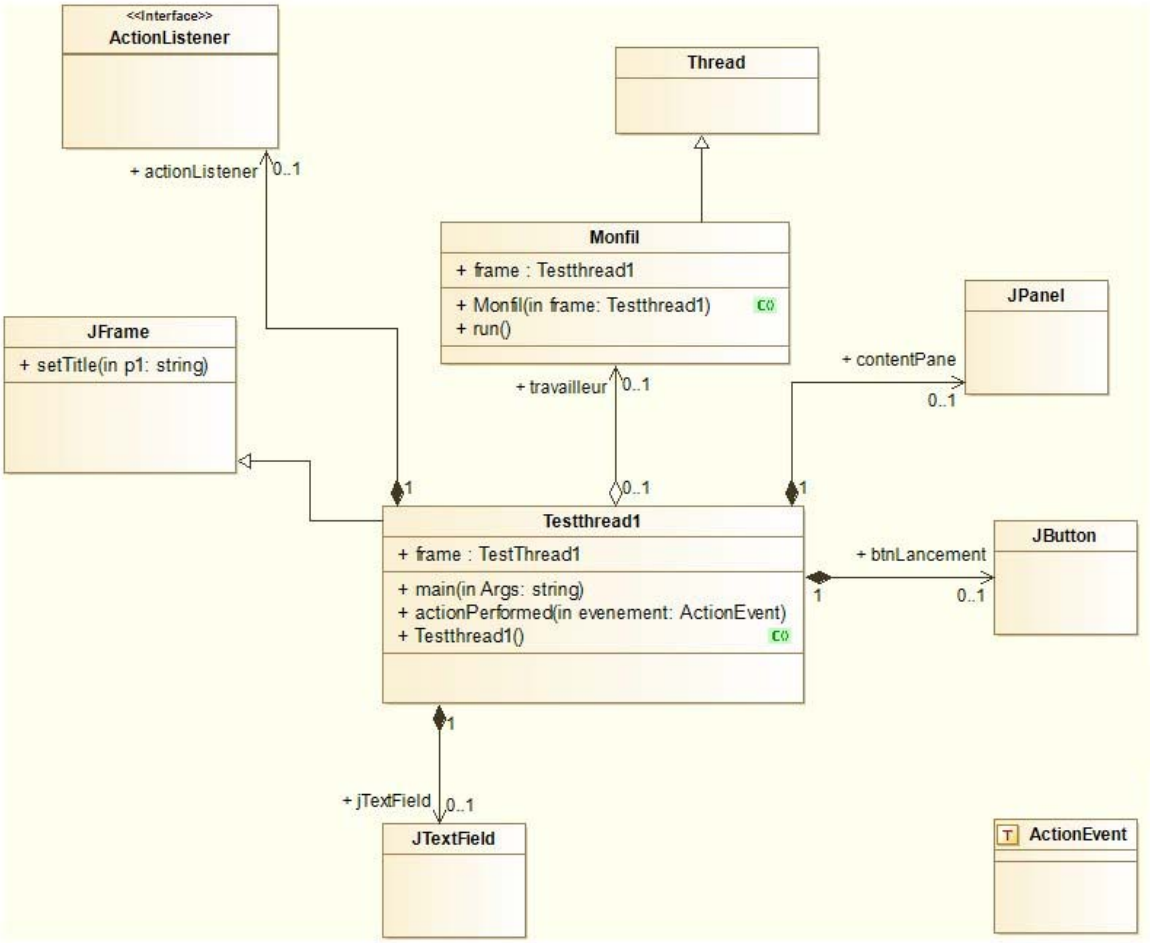
```
Class monthread  extends Threads
{

Public void run()
{
// code de la tache
}
Public static void main (String args[]){
monthread t= new monthread() ; // création de l'objet thread avec le constructeur par défaut
t.start()// execution du thread et de sa méthode run
}

}
```

Une fois que vous avez **invoqué la méthode start, le thread devient Exécutable**. Entre autre chose, **la méthode run sera alors automatiquement appelée**. (Il ne faut surtout pas appeler directement la méthode run). Chaque tâche doit se trouver dans la méthode run. Le thread est alors exécutable et il le reste jusqu'à la fin de la méthode run. Lorsque cette dernière est complètement exécutée, le thread meurt de lui-même.

Reprendre le projet précédent en adoptant le concept de thread (le diagramme de classe est donné ci-dessous).
On créera une classe Monfil dans laquelle on implémentera la méthode run de tout à l'heure.
Le thread nommé travailleur sera créé à chaque appui sur le bouton (dans la méthode action performed)



Code source :

```
import java.awt.BorderLayout;
import java.awt.EventQueue;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;
import javax.swing.JTextField;
import java.awt.event.*;

public class testthread1 extends JFrame implements ActionListener  {

	private JPanel contentPane;
	private final JButton btnLancement = new JButton("lancement");
	private JTextField textField;
	private  static testthread1 frame;
	/**
	 * Launch the application.
	 */
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {
			public void run() {
				try {
					frame = new testthread1();
					frame.setVisible(true);

				} catch (Exception e) {
					e.printStackTrace();
				}
			}
		});
	}

	/**
	 * Create the frame.
	 */
	public testthread1() {
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		setBounds(100, 100, 248, 173);
		contentPane = new JPanel();
		contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
		setContentPane(contentPane);
		contentPane.setLayout(null);
		btnLancement.setBounds(55, 11, 120, 31);
		contentPane.add(btnLancement);
		btnLancement.addActionListener(this);
		textField = new JTextField();
		textField.setBounds(55, 68, 120, 31);
		contentPane.add(textField);
		textField.setColumns(10);
	}
	public void actionPerformed(ActionEvent evenement) {
```

```
		-----------------------------------------------------// création du thread
		-----------------------------------------------------// lancement du thread
```

```
    }
}
```

```
Place ton code ici de la classe monfil
```

L'interface runnable

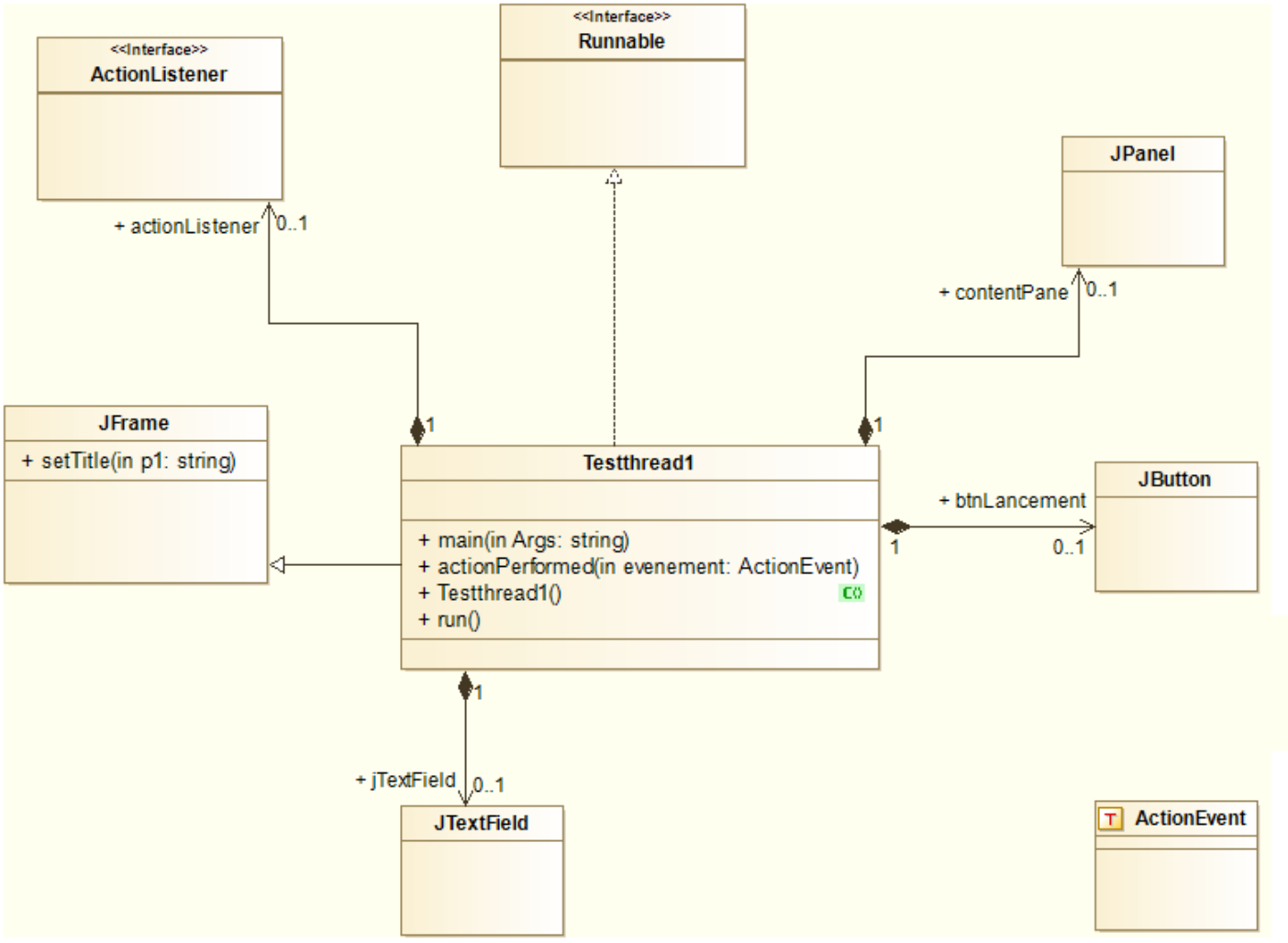
On vient de voir comment créer des threads à partir de la classe Thread. Le problème c'est qu'un objet thread ne peut dériver que de la classe thread et de rien d'autre (Java ne possède pas d'héritage multiple). Nous ne pouvons pas hériter de deux classes parents, et nous devons donc passer par une interface qui s'appelle **Runnable**. Cette interface dispose de la méthode run que vous devez absolument redéfinir (principe même d'une interface).

```
Class monthread  extends autreclasse  implements Runnable
{

Public void run()
{
// code de la tache
}
Public static void main (String args[]){
monthread t= new monthread() ; // création de l'objet thread avec le constructeur par défaut
new Thread (t).start()// execution du thread et de sa méthode run
}

}
```

Reprendre le projet précédent en adoptant le concept de thread et d'interface Runnable (le diagramme de classe est donné ci-dessous).



Les sockets et la programmation réseau

1. Introduction

Les échanges avec le réseau sont devenus omniprésents dans les applications et entre les applications. Ils permettent notamment :

- un accès a des serveurs comme une base de données
- d'invoquer des services distants
- de développer des applications web
- d'échanger des données entres applications

La plupart de ces fonctionnalités sont mises en œuvre via des API de haut niveau mais celles-ci utilisent généralement des API de bas niveau pour interagir avec le réseau.

Depuis son origine, Java fournit plusieurs classes et interfaces destinées à faciliter l'utilisation du réseau par programmation en reposant sur les sockets. Celles-ci peuvent être mises en œuvre pour réaliser des échanges utilisant le protocole réseau IP avec les protocoles de transport TCP ou UDP. Les échanges se font entre deux parties : un client et un serveur.

2. La classe InetAddress

Cette classe permet de manipuler les adresses Internet.

Soit le programme suivant :

```
import java.net.*;
import java.util.Scanner;

public class Testnet {

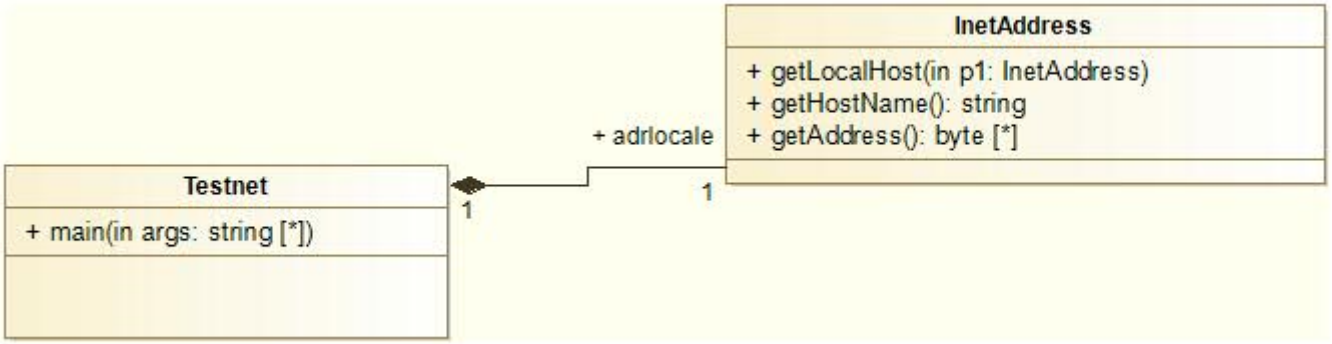

	public static void main(String[] args) {

		byte IP[] = new byte [4];

		try
		{
		InetAddress adrlocale = InetAddress.getLocalHost();
		String hostname = adrlocale.getHostName();
		IP=adrlocale.getAddress();
		System.out.printf("%x %x %x %x \n",IP[0],IP[1],IP[2],IP[3]);
		System.out.println("adresse locale ="+adrlocale.getHostAddress() );
		System.out.println("nom de la machine ="+hostname );
		}catch (Exception e)
		{
			e.printStackTrace();
		}
	}
}
```

Testez ce programme. En analysant ce programme, complétez le nom de chaque méthode dans le tableau ci-après :

La classe InetAddress	
	Retourne un objet contenant l'adresse Internet de la machine locale.
	Retourne le nom de la machine
	Retourne l'adresse internet sous forme d'un tableau de 4 octets



3. L'utilisation du protocole TCP

TCP est un protocole qui permet une connexion de type point à point entre deux applications. C'est un protocole fiable qui garantit la réception des données. En contrepartie, ce protocole offre de moins bonnes performances mais c'est le prix à payer pour la fiabilité.

TCP utilise la notion de port pour permettre à plusieurs applications d'exploiter ce même protocole. Dans une liaison entre deux ordinateurs, l'un des deux joue le rôle de serveur et l'autre celui de client.

3.1. Mise en place d'un client (la classe Socket)

Le concept de socket est assez difficile à appréhender. Une socket permet d'accomplir les communications inter-processus (IPC). Cela veut dire qu'un socket est utilisé pour permettre aux processus de communiquer entre eux de la même manière que le téléphone nous permet de communiquer entre nous.

Les sockets implémentent le protocole TCP (Transmission Control Protocol). La classe contient les méthodes de création des flux d'entrée et de sortie correspondants. Les sockets constituent la base des communications par le réseau.

Comme les flux Java sont transformés en format TCP/IP, il est possible de communiquer avec l'ensemble des ordinateurs qui utilisent ce même protocole. La seule chose importante au niveau du système d'exploitation est qu'il soit capable de gérer ce protocole.

Cette classe encapsule la connexion à une machine distante via le réseau. Elle gère la connexion, l'envoi de données, la réception de données et la déconnexion.

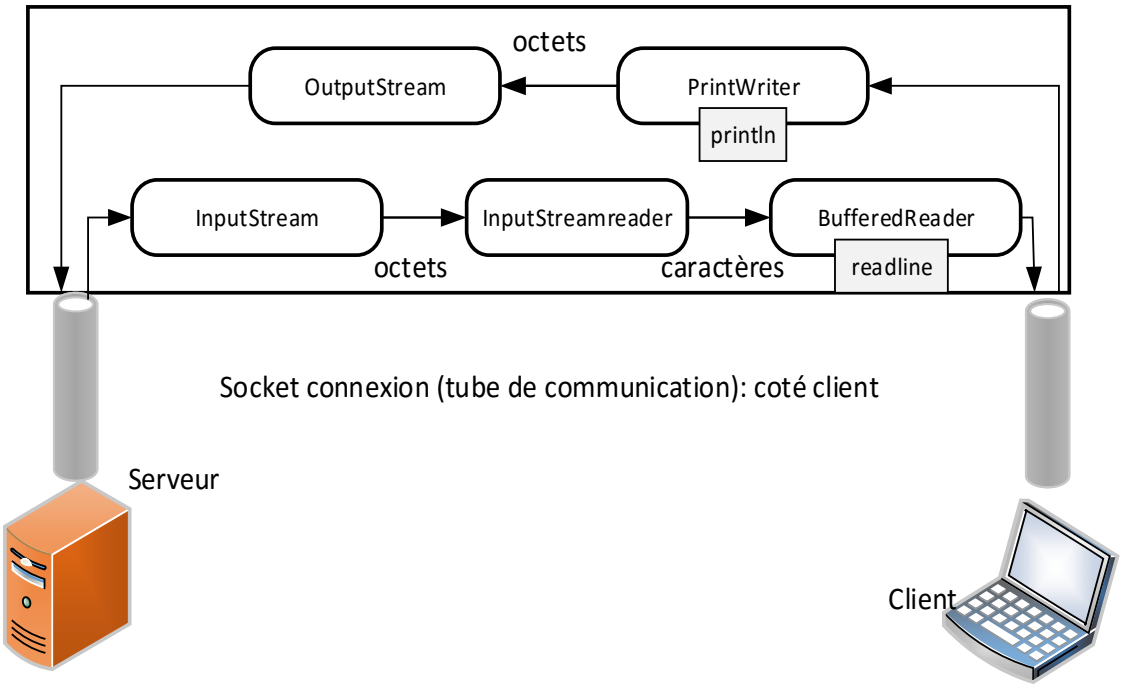
La classe Socket possède plusieurs constructeurs dont les principaux sont :

Constructeur	Rôle
Socket()	Constructeur par défaut
Socket(String nom, int port)	Crée une socket et la connecte à un port de l'ordinateur distant.

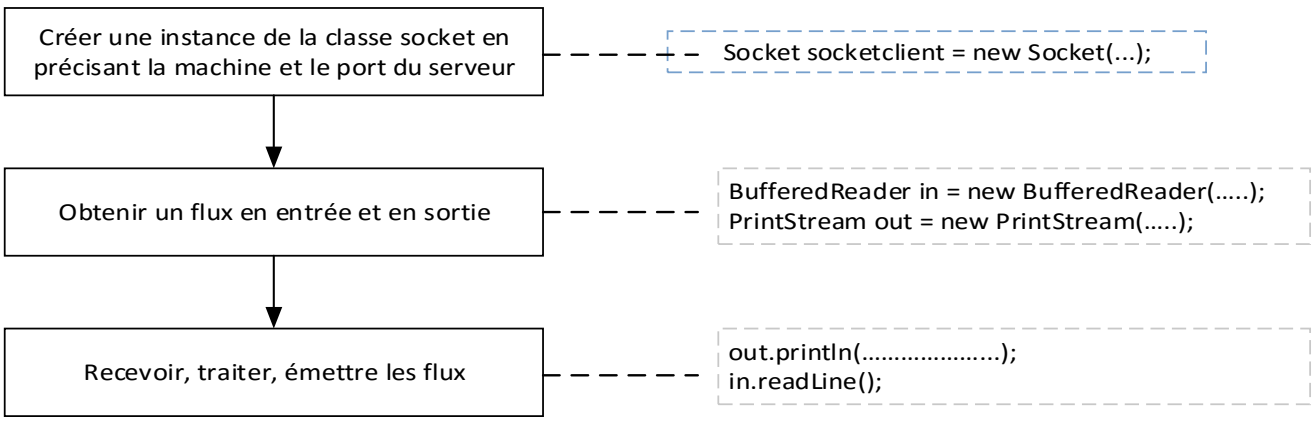
La classe Socket possède de nombreuses méthodes dont les principales sont :

Méthode	Rôle
InetAddress getinetAddressQ	Renvoie l'adresse l.P. à laquelle la socket est connectée
void close()	Fermer la socket
InputStream getinputStream()	Renvoie un flux d'octets en entrée pour recevoir les données de la socket
OutputStream getOutputStream()	envoie un flux d'octets en sortie pour émettre les données de la socket
int getPort()	Renvoie le port utilisé par la socket

Les flux d'informations gérées par la socket sont modélisées ci-dessous :



La démarche pour créer un client reste toujours la même :



On donne le programme d'un client en JAVA :

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

public class TestClientTcp {
        final static int port = ……;
        public static void main(String[] args) {

                Socket socket; DataInputStream userinput;
                PrintStream theOutputStream;

                try {

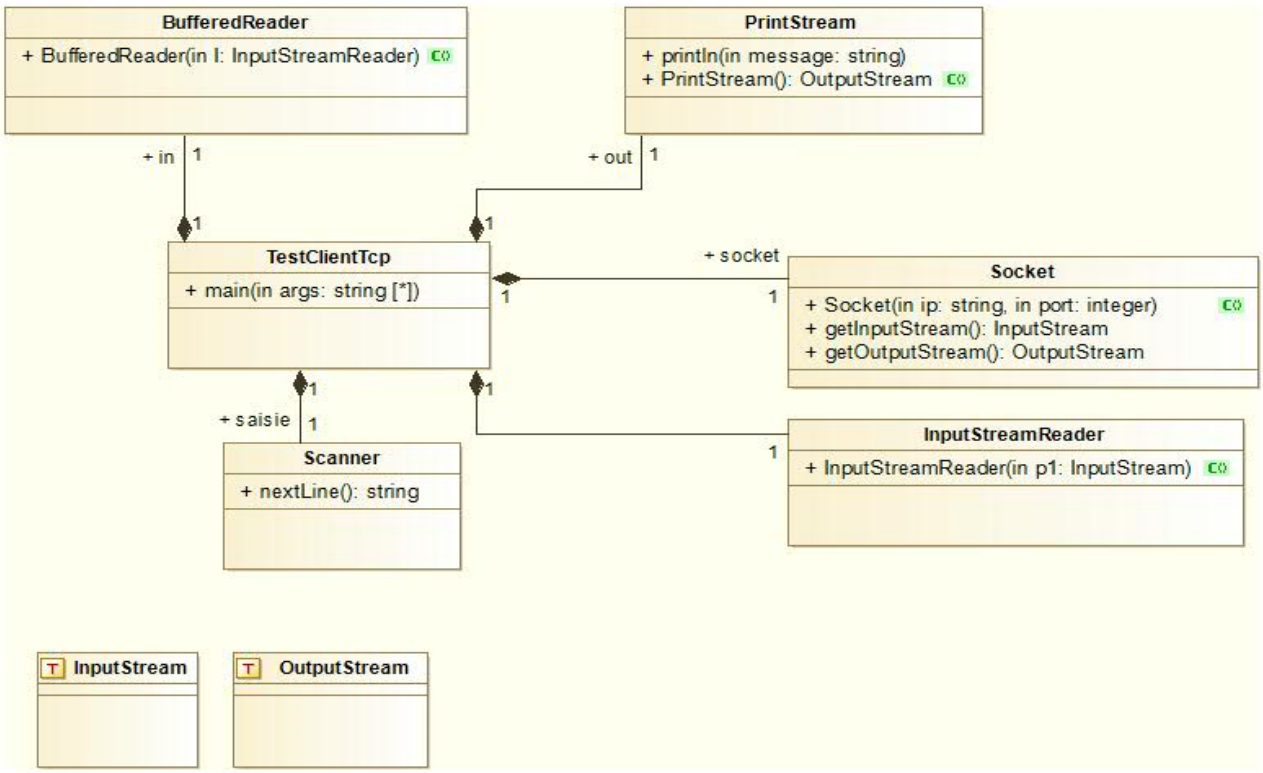
                socket = new Socket(……);

                BufferedReader in = new BufferedReader(new InputStreamReader(socket._____));
                PrintStream out = new PrintStream(socket._____);
                Scanner saisie= new Scanner(System.in);
                String message= saisie.nextLine();

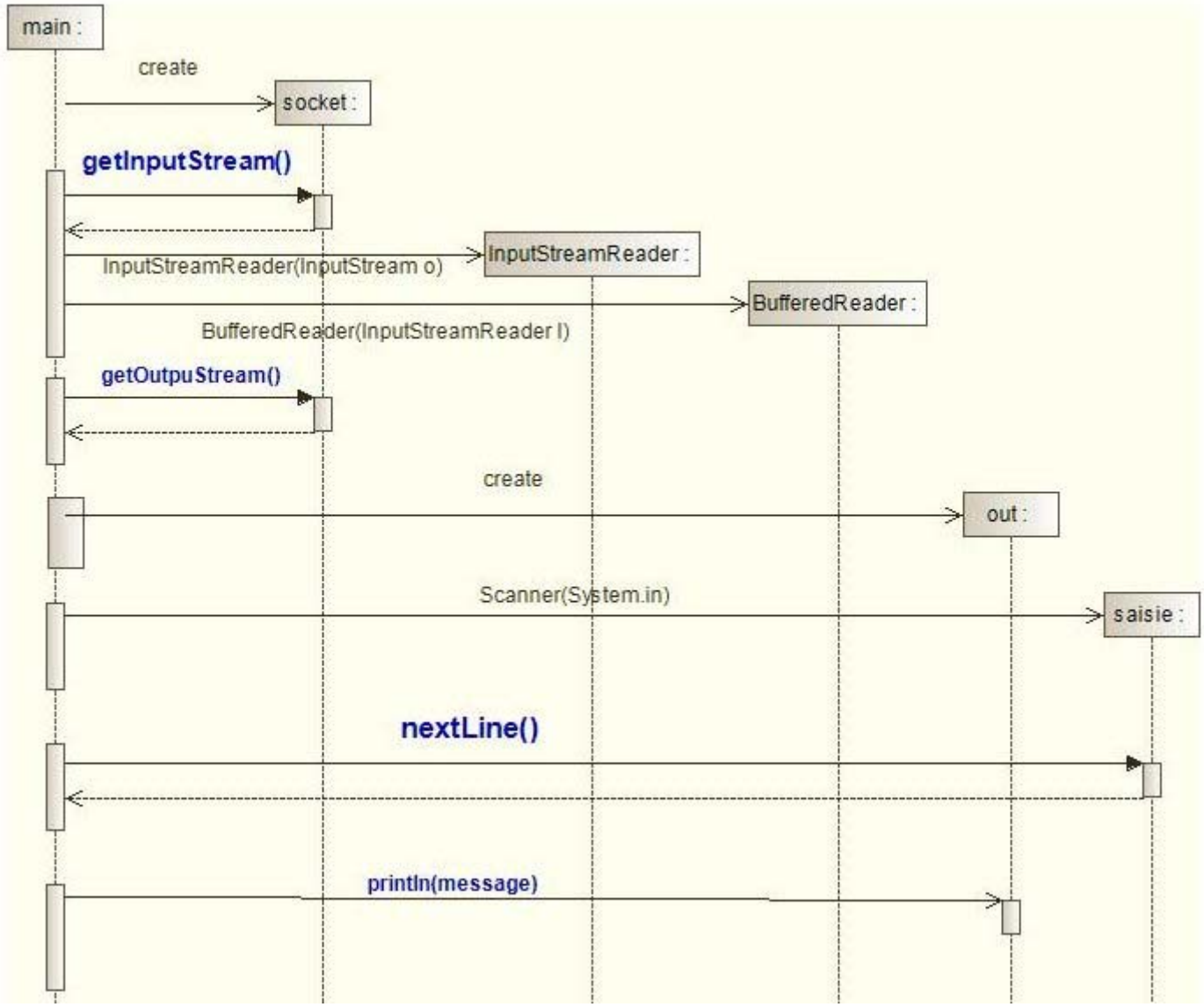
                out.println(_____);
                System.out.println(in._____);
                }
                catch (Exception e) {
                e.printStackTrace();}
        }

}
```

Son diagramme de classe



Le diagramme de séquence



On souhaite implanter ce client sur la raspberry afin de dialoguer avec un serveur windows (machine ou se trouve votre suite eclipse).

Complétez le programme et testez le sur la raspberry (le programme serveur est baptisé serveur.exe).

Relevez les trames sur wireshark et relevez les échanges entre le client et le serveur.

3.2. Mise en place d'un serveur (la classe ServerSocket)

Maintenant que nous avons implémenté un client simple qui reçoit et émet des données vers un serveur, intéressons-nous à la fabrication d'un serveur simple qui devra envoyer des informations et recevoir des requêtes d'un client. Une fois que ce serveur (service) sera lancé, il devra attendre qu'un client se connecte à l'un de ces ports. Nous choisissons le port 1200, qui n'est utilisé par aucun service standard.

La classe ServerSocket est utilisée côté serveur. Elle attend simplement les connexions du ou des clients. C'est un objet du type Socket qui prend en charge la transmission des données.

Cette classe représente la partie serveur du socket. Un objet de cette classe est associé à un port sur lequel il va attendre les connexions d'un client. Généralement, à l'arrivée d'une demande de connexion, un thread est lancé pour assurer le dialogue avec le client sans bloquer les connexions des autres clients.

La classe SocketServer possède plusieurs constructeurs dont les principaux sont :

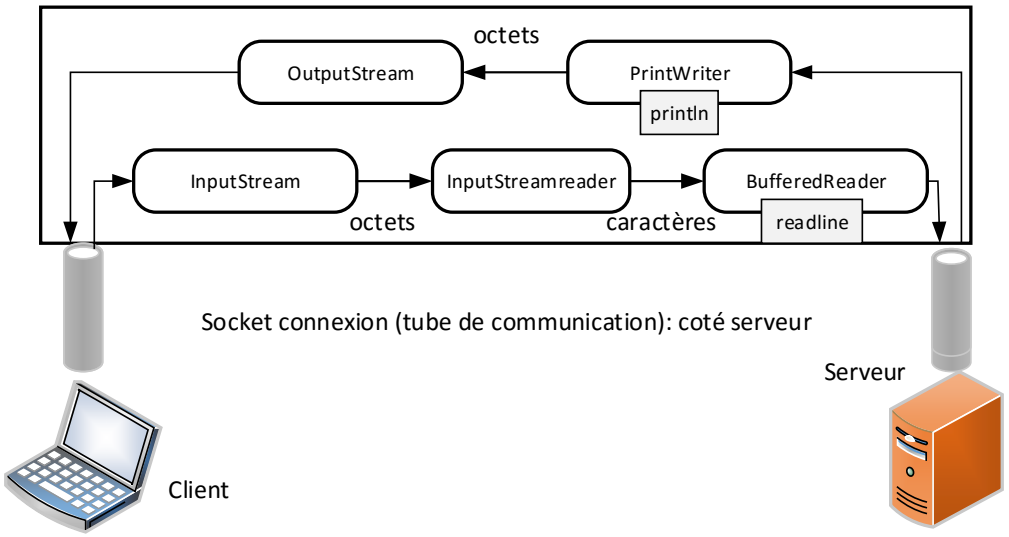
Constructeur	Rôle
ServerSocketO	Constructeur par défaut
ServerSocket(int port)	Créer une socket sur le port fourni en paramètre
ServerSocket(int port, int file)	Créer une socket sur le port et avec la taille maximale de la file fournis en paramètres

Tous ces constructeurs peuvent lever une exception de type IOException.

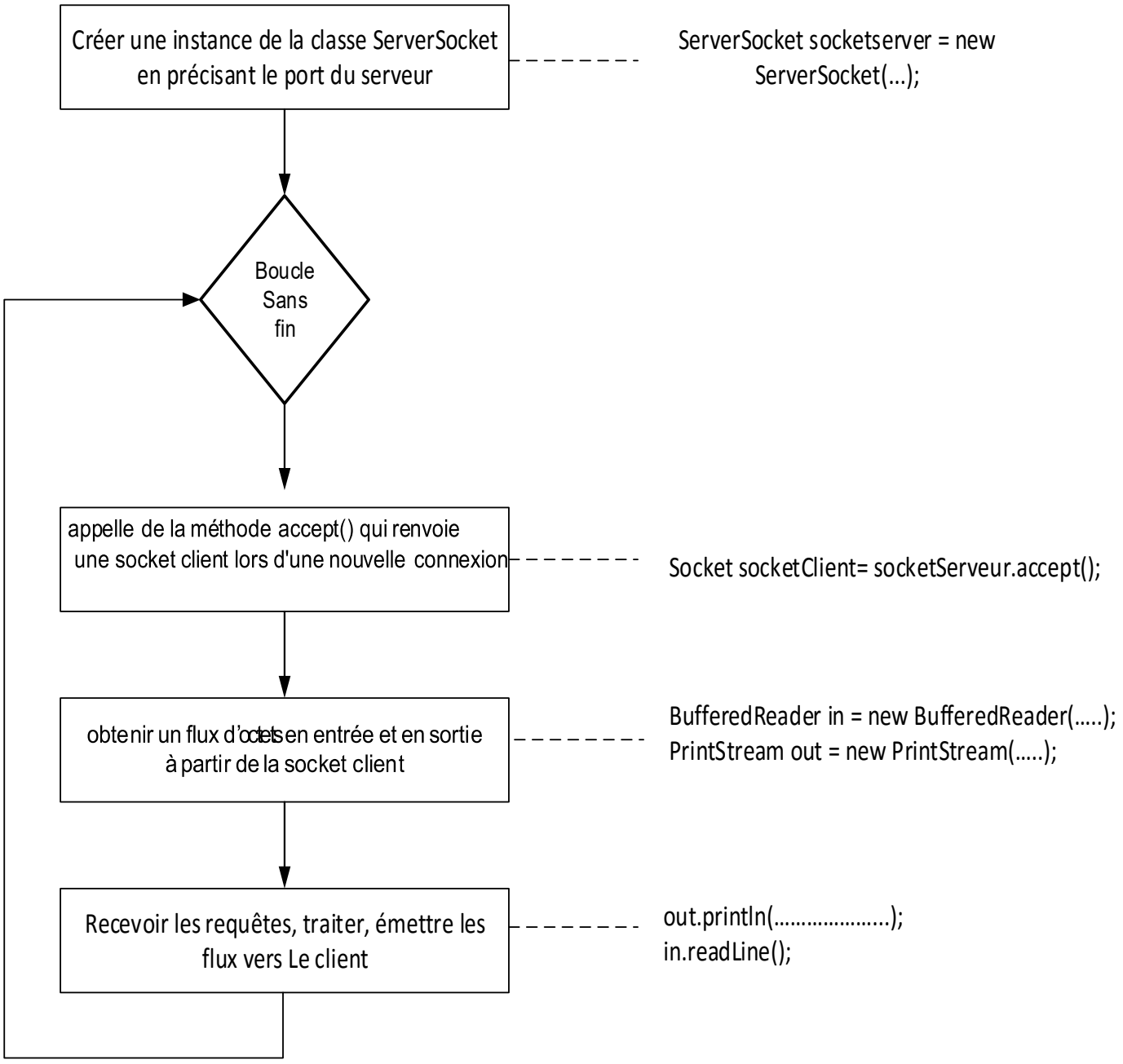
La classe SocketServer possède plusieurs méthodes :

Méthode	Rôle
Socket accept()	Attendre une nouvelle connexion d'un client (function bloquante)
void close()	Fermer la socket

Les flux d'informations gérées par le serveur sont modélisés ci-dessous :



La démarche pour créer un serveur reste toujours la même :

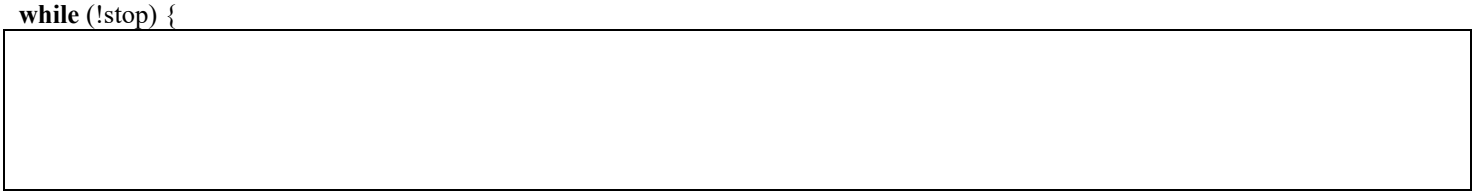


Le programme serveur est donné ci-dessous :

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

public class TestServeurTCP {
        final static int port = 1200;

        public static void main (String[] args) {
        try
        {
        ServerSocket socketServeur = new ServerSocket(port);
        System.out.println ("Lancement du serveur") ;
                Socket socketClient= socketServeur.accept();
                String message = "ok";
                System.out.println ("Connexion avec : "+socketClient.getInetAddress ());
                BufferedReader in = new BufferedReader(new InputStreamReader (socketClient.getInputStream( )));
                PrintStream pw = new PrintStream(socketClient.getOutputStream());
                System.out.println("je suis ici");
                boolean stop=false;
while (!stop) {
```



```
                        }

                }
                pw.close();
                in.close();
                socketClient.close();

        }
        catch (Exception e) { e.printStackTrace();}
        }
        }
```

La partie service et son code associé est donnée dans la boucle :

```
while (!stop) {

// code du service

}
```

Ce service doit permettre :

- ✓ de recevoir le message du client
- ✓ Afficher le message envoyé par le client (si le message est non nul)
- ✓ De renvoyer à l'identique le message reçu vers le client (si le message est non nul)
- ✓ Si le message reçu est « ok » de positionner la variable stop à true.

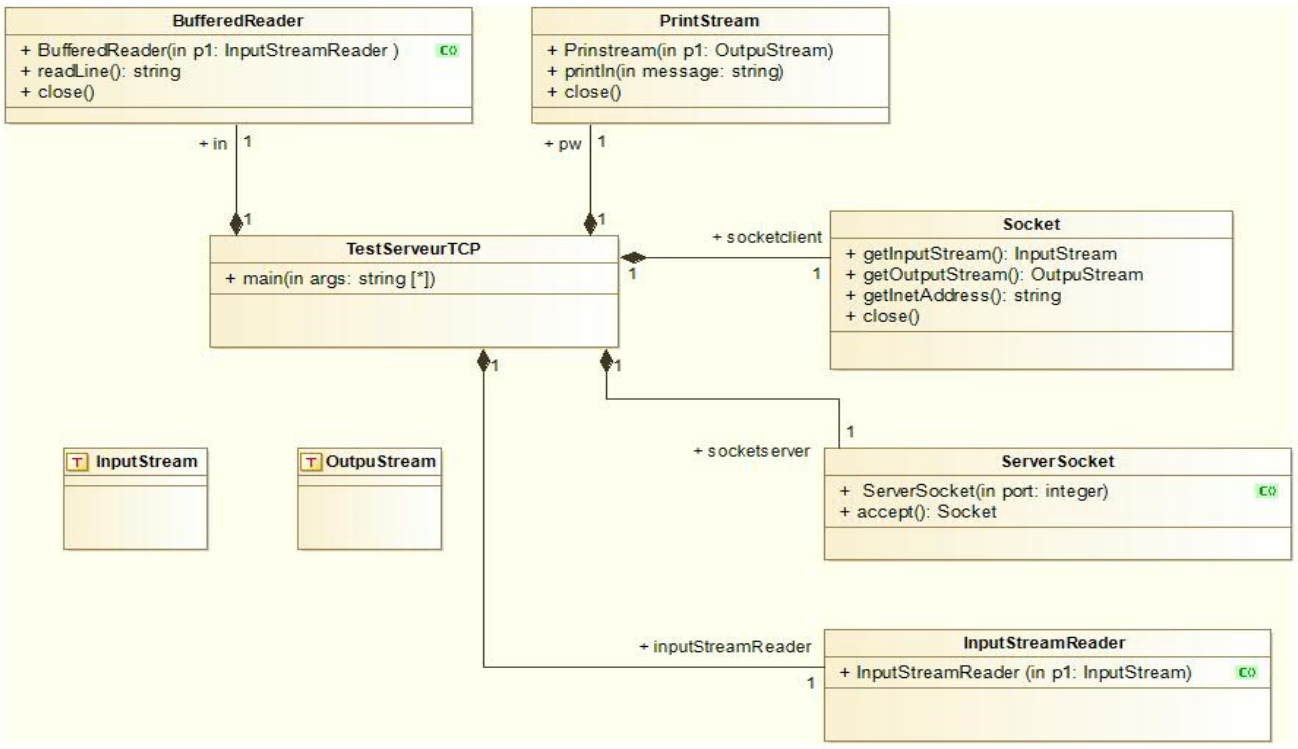
On donne ci-dessous le diagramme de classe et de séquence

Complétez le programme en proposant le code de ce service. Testez le sur la raspberry (le programme client est baptisé client.exe).
Relevez les trames sur wireshark et relevez les échanges entre le client et le serveur.

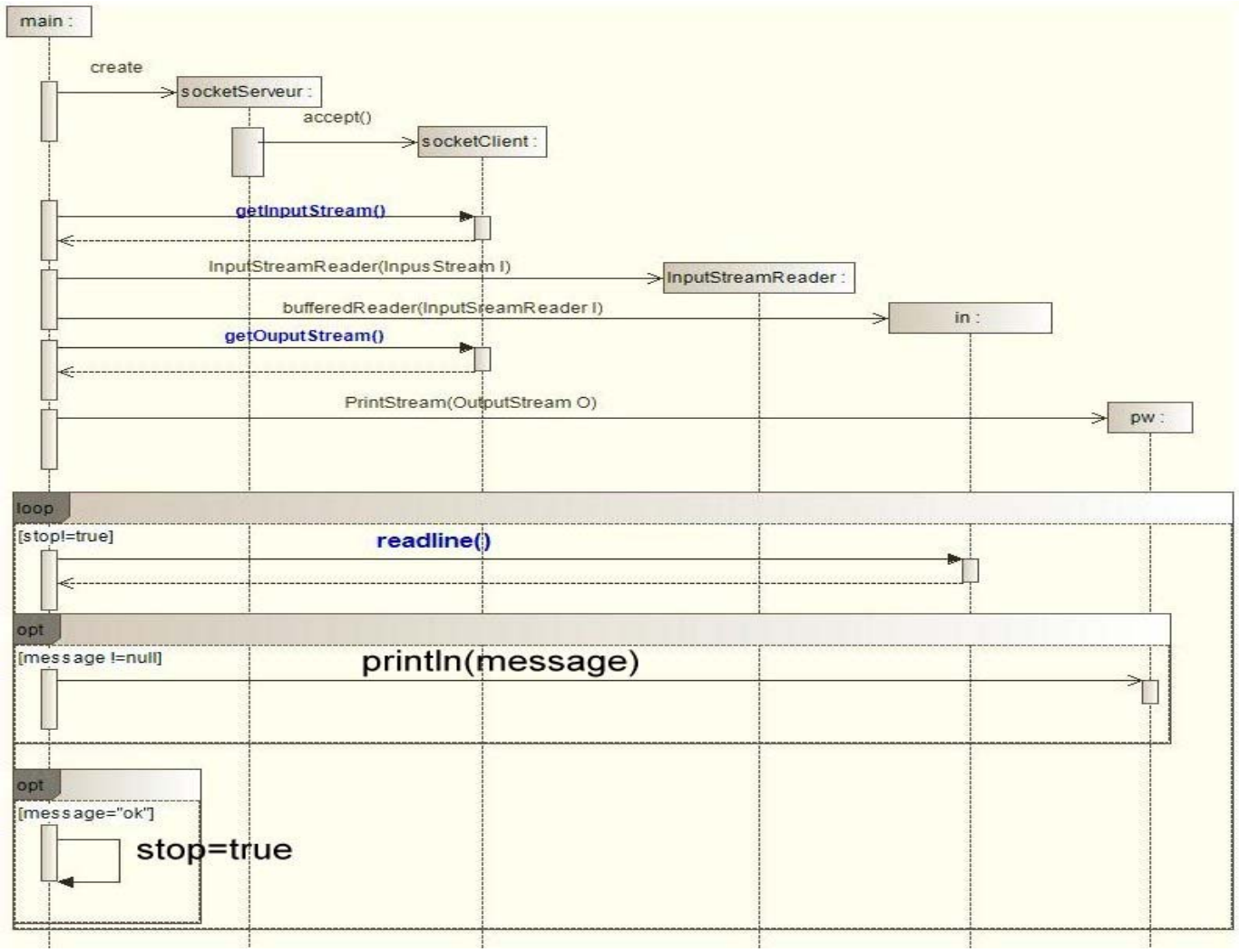
Pendant la communication du premier client vers le serveur, lancez un deuxième client que constatez-vous ? **(Attention aux sécurités JAVA)**

✍

Le diagramme de classe



Le diagramme de séquence



Code source :

4. L'utilisation du protocole UDP (user datagram protocol)

4.1. La classe DatagramSocket

UDP est un protocole basé sur IP qui permet une connexion de type point à point ou de type multipoint. C'est un protocole qui ne garantit pas que les données arrivent dans l'ordre d'émission. En contrepartie, ce protocole offre de bonnes performances car il est très rapide mais à réserver à des tâches peu importantes.

Pour assurer les échanges, UDP utilise la notion de port, ce qui permet à plusieurs applications d'utiliser UDP sans que les échanges interfèrent les uns avec les autres. Cette notion est similaire à la notion de port utilisée par TCP.
UDP est utilisé dans de nombreux services "standards" tels que echo (port 7), DayTime (13), etc ...
L'échange de données avec UDP se fait avec deux sockets, l'une sur le serveur, l'autre sur le client. Chaque socket est caractérisé par une adresse internet et un port.

Pour utiliser le protocole UDP, JAVA définit deux classes DatagramSocket et DatagramPacket
Cette classe crée un Socket qui utilise le protocole UDP (Unreliable Datagram Protocol) pour émettre ou recevoir des données.

Cette classe possède plusieurs constructeurs :

Constructeur	Rôle
DatagramSocket()	Créer une socket attachée à toutes les adresses IP de la machine et à un des ports libres sur la machine
DatagramSocket(int port)	Créer une socket attachée à toutes les adresses IP de la machine et au port précisé en paramètre
DatagramSocket(int port, InetAddress address)	Créer une socket attachée à l'adresse IP et au port précisé en paramètres

Tous les constructeurs peuvent lever une exception de type SocketException : en particulier, si le port précisé est déjà utilisé lors de l'instanciation de l'objet DatagramSocket, une exception de type BindException est levée. Cette exception hérite de SocketException.

La classe DatagramSocket définit plusieurs méthodes :

Méthode	Rôle
Close()	Fermer la socket et ainsi libérer le port
receive(DatagramPacket IN)	Recevoir des données
send(DatagramPacket OUT)	Envoyer des données
int getPort()	Renvoyer le port associé à la socket
void setSoTimeout(int durée)	Préciser un timeout d'attente pour la réception d'un message.

Par défaut, un objet Datagram.Socket ne possède pas de timeout lors de l'utilisation de la méthode receive(). La méthode bloque donc l'exécution de l'application jusqu'à la réception d'un packet de données. La méthode setSoTimeoutO permet de préciser un timeout en millisecondes. Une fois ce délai écoulé sans réception d'un paquet de données, la méthode lève une exception du type SocketTimeoutException.

La classe DatagramPacket encapsule une adresse internet, un port et les données qui sont échangées grâce à un objet de type **DatagramSocket.** Elle possède plusieurs constructeurs pour encapsuler des paquets émis ou reçus.

Constructeur	**Rôle**
DatagramPacket(byte tampon[], int taille)	Encapsule des paquets en réception dans un tampon
DatagramPacket(byte port[], int taille, InetAddress adresse, int port)	Encapsule des paquets en émission à destination d'une machine

Cette classe propose plusieurs méthodes pour obtenir ou mettre à jour les informations sur le paquet encapsulé.

Méthode	**Rôle**
InetAddress getAddress ()	Renvoyer l'adresse du serveur
byte[] getData()	Renvoyer les données contenues dans le paquet
int getPort ()	Renvoyer le numéro de port
int getLength ()	Renvoyer la taille des données contenues dans le paquet
setData(byte[] tableau)	Mettre à jour les données contenues dans le paquet

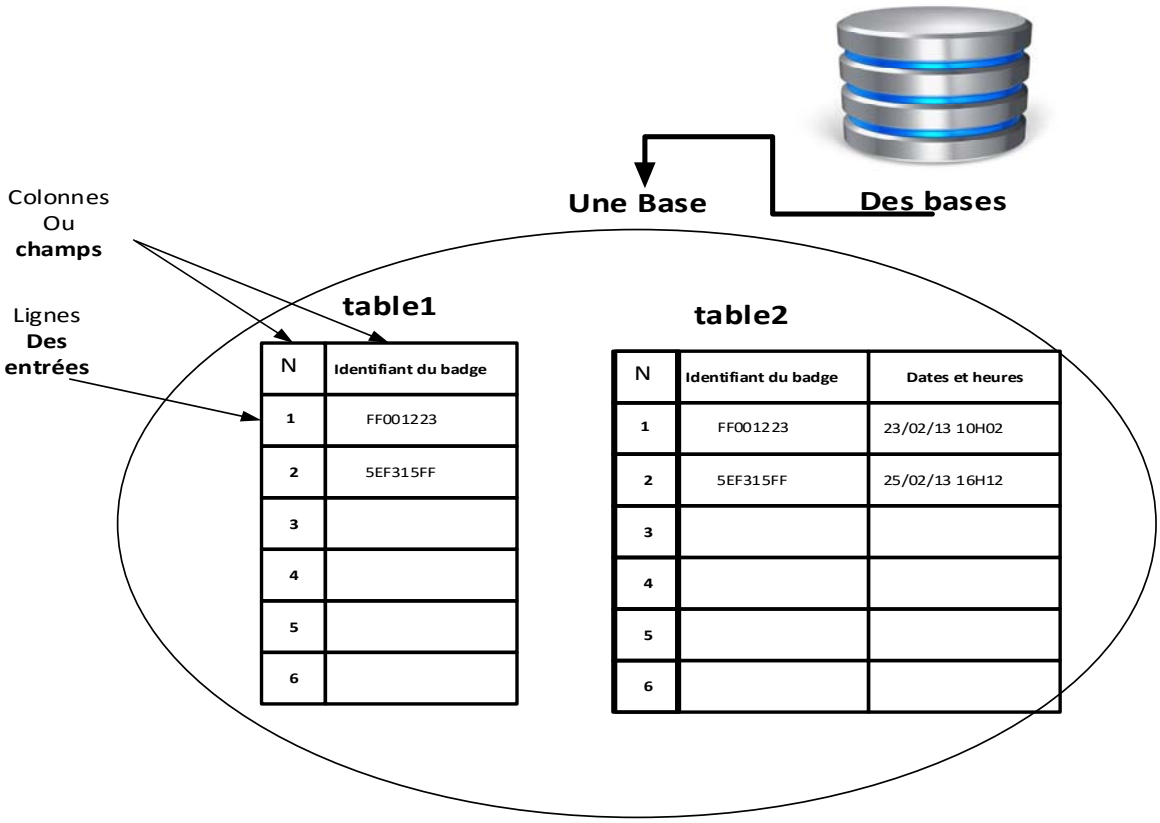
Initiation aux bases de données relationnels

1. Définition

Une base de données (**BDD**) permet d'enregistrer des données de façon organisée et hiérarchisée. La base de données est stockée dans un fichier sur un disque dur et les informations y sont rangées de manière très particulière.
L'organisation des données au sein des Systèmes de Gestion des Bases de Données Relationnelles (**SGBDR** ou RDBMS en anglais). Les **SGBD** sont les programmes qui se chargent tout simplement du stockage de vos données. Fondée sur une représentation logique des données en respectant le modèle relationnel, les bases de données **relationnelles** ont su s'imposer car elles s'appuient toutes sur le même langage standardisé et normalisé qu'est le **SQL (Structured Query Language)**.

2. Le vocabulaire

Dans un serveur de données cohabite plusieurs bases de données.
La base de données contient plusieurs tables (on peut en mettre autant que l'on veut à l'intérieur). Chaque table est en fait un tableau où les colonnes sont appelées champs et où les lignes sont appelées entrées.



3. Le langage SQL

SQL signifie (Structured Query Language) c'est-à-dire Langage d'interrogation structuré. SQL est un langage complet de gestion de bases de données relationnelles.
Il a été conçu par IBM dans les années 70. Il est devenu le langage standard des systèmes de gestion de bases de données (SGBD) relationnelles (SGBDR)., c'est à la fois :

- un langage d'interrogation de la base (ordre SELECT)
- un langage de manipulation des données (LMD; ordres UPDATE, INSERT, DELETE)
- un langage de définition des données (LDD ; ordres CREATE, ALTER, DROP),
- un langage de contrôle de l'accès aux données (LCD ; ordres GRANT, REVOKE).

Le langage SQL est utilisé par les principaux SGBDR: DB2, Oracle, Informix, Ingres, RDB,... Chacun de ces SGBDR a cependant sa propre variante du langage.

Le SGBDR utilisé par la suite sera du type MYSQL, afin d'utiliser le logiciel

4. JAVA et le langage SQL

Avant d'aborder par un exemple type l'utilisation du langage SQL on va définir les types de données sous SQL

SQL dispose de ses propres types de données

Type SQL	Description	Type Java conseillé	Méthode Java
BIT	1 bit	boolean	getBoolean
TINYINT	entier 8 bits	byte	getByte
SMALLINT	entier 16 bits	short	getShort
INTEGER	entier 32 bits	int	getInt
BIGINT	entier 64 bits	long	getLong
REAL	flottant 32 bits	float	getFloat
DOUBLE	flottant 64 bits	double	getDouble
CHAR(n)	chaîne d'exactement *n* caractères	String	getString
VARCHAR(n)	chaîne d'au plus *n* caractères	String	getString
DATE	date	Java.sql.Date	getDate
TIME	heure	Java.sql.Time	getTime
TIMESTAMP	date et heure	Java.sql.Timestamp	getTimestamp
DECIMAL (c, d)	nombre décimal de *c* chiffres dont *d* décimales représenté de façon exacte	java.math.BigDecimal	getBigDecimal

On verra lors de la création de la base de données que chaque champ dans la table a son propre type de données.
Au préalable il nous faut Installer le driver JDBC pour MySQL sous Eclipse. Il nous faut aussi installer le logiciel WAMPSERVER et mettre en place la base de données (voir annexe du TP N°5). Nous étudierons que ce qui est nécessaire au TP5, pour connaitre le langage SQL de manière plus poussée on trouvera beaucoup d'ouvrages spécialisés dans ce domaine

4.1. Connexion à la base de données

La première chose à faire est d'établir une connexion avec votre SGBD MYSQL vers la base de données. Cette base de données aura le format suivant :

La table sera baptisée **boisson**

Nom_boisson	identifiant	prix
cafe	F12	3.2
limonade	F8	4.1

Pour qu'une application JAVA puisse accéder à une base de données, il sufiit que le SGBDR fournisse ce que l'on nomme un pilote JDBC. Il s'agit d'un ensemble de classes JAVA qui vont permettre l'utilisation du SGBD, par le biais de requêtes SQL

Avec le driver JDBC pour MySQL, le nom de la classe qui l'implémente est "com.mysql.jdbc.Driver.class"

Ainsi, vous devrez charger le pilote avec cette ligne de code :

```
Class.forName("com.mysql.jdbc.Driver");
```

On n'a pas besoin de créer une instance du pilote et de le référencer à DriverManager.

Pour établir une connexion pour se connecter à votre SGBD on donne la ligne de code suivante :

```
Connection con =DriverManager.getConnection(url,"MonLogin","MonMotDePasse");
```

Exemple :

```
String url = "jdbc:mysql://localhost:3306/boisson";
conn = DriverManager.getConnection(url, "root", "root");
```

On peut aussi utiliser une adresse IP

```
String url = "jdbc:mysql://127.0.0.1 :3306/boisson » ;
conn = DriverManager.getConnection(url, "root", "root1") ;
```

root est le nom d'utilisateur de la base de données
root1 le mot de passe

4.2. La création de la table

Un objet **Statement** est ce que votre instruction SQL envoie vers le SGBD. Vous créerez simplement un objet Statement puis, l'exécuterez, lui fournissant la méthode d'exécution appropriée avec l'instruction SQL que vous voulez envoyer. Pour une instruction SELECT, la méthode à utiliser est **executeQuery**. Pour les instructions visant à créer ou modifier des tables, la méthode est **executeUpdate.**

Vous devez avoir l'instance d'une connexion active pour créer un objet Statement. Dans l'exemple suivant, nous utilisons notre objet Connection conn, pour créer l'objet Statement stmt :

```
Statement stmt = conn.createStatement ();
```

L'instruction SQL qui suit crée la table boisson. Les entrées entre les parenthèses sont le nom de chacune des colonnes suivit d'un espace puis du type de données que la colonne peut recevoir. Une virgule sépare chacune des colonnes les unes des autres. Le type VARCHAR est créé avec une longueur maximum de 32 caractères.

```
CREATE TABLE boisson (Nom_boisson  VARCHAR(32),identifiant VARCHAR(10), prix FLOATR)
```

String creerTableboisson = "CREATE TABLE boisson (Nom_boisson VARCHAR(32),identifiant VARCHAR(10), prix FLOATR) "

On va donc utiliser la méthode executeUpdate afin de créer cette table :

```
stmt.executeUpdate(creerTableboisson) ;
```

4.3. Modifiez ou créez un champ dans la base de données

Les instructions consistants à créer des tables, modifier des tables ou effacer des tables sont des exemples d'instructions **DDL(Data Definition Language)** et sont exécuter avec la méthode **executeUpdate.** La méthode executeUpdate est utilisée pour exécuter les instructions SQL qui mettent à jour une table. En pratique, executeUpdate est utilisé le plus souvent pour mettre des tables à jour plutôt que de les créer

. Nous allons entrer nos données dans une table une ligne à la fois, fournissant l'information à stocker dans chacune des colonnes de cette ligne. Notez que les valeurs insérées dans les colonnes doivent être dans le même ordre que les colonnes à leur création.

```
stmt.executeUpdate("INSERT INTO boisson VALUES ('cafe','F12', 3.2)");
```

```
stmt.executeUpdate("INSERT INTO boisson VALUES ('limonade','F8', 4.1)");
```

4.4. Accédez aux données d'une table

L'instruction **SELECT** permet d'accéder à des valeurs dans la table. L'étoile (*) dans l'instruction SQL indique que toutes les colonnes devront être sélectionnées. N'ayant pas de clause WHERE définissant la restriction, nous effectuons la sélection sur toute la table.

```
SELECT * FROM boisson
```

La base de données va vous répondre en fournissant tous les éléments suivants :

Nom_boisson	identifiant	prix
Cafe	F12	3.2
Limonade	F8	4.1

stmt.executeQuery("SELECT Nom_boisson, prix FROM boison");

La base de données va vous répondre en fournissant tous les éléments suivants :

Nom_boisson	prix
Cafe	3.2
Limonade	4.1

Mais comment récupérer le résultat renvoyé par la base de données. Grâce à un objet résultat.

JDBC renvoie les résultats dans un **objet ResultSet**, donc, nous avons besoin de déclarer une instance de la class ResultSet pour contenir nos résultats. Le code qui suit déclare l'objet ResultSet resultat et lui assigne le résultat de notre requête précédente :

ResultSet resultat = stmt.executeQuery("SELECT Nom_boisson, prix FROM boison");

Pour pouvoir accéder à leurs noms et à leurs prix, nous irons dans chaque ligne et récupèrerons les valeurs selon leur type. La méthode next déplace ce qui est appelé le curseur à la ligne suivante, et fait de cette ligne (appelée la ligne courante) celle sur laquelle nous travaillons. Initialement, ce curseur est positionné juste au dessus de la première ligne d'un objet ResultSet, donc le premier appel à la méthode next déplace le curseur à la première ligne et en fait la ligne courante. L'invocation successive de la méthode next déplace le curseur vers le bas, ligne par ligne.

Nous utilisons la méthode **getXXX** du type approprié pour accéder à une valeur dans chaque colonne. Pour la requête précédente la première colonne de chaque ligne du resultset est **Nom_boisson**, qui contient une valeur de type SQL VARCHAR. La méthode pour accéder à une valeur de type VARCHAR est **getString**. La seconde colonne de chaque ligne contient une valeur de type **SQL FLOAT**, et la méthode pour accéder à ces valeurs est **getFloat**. Le code qui suit accède aux valeurs contenues dans la ligne courante du resultset et affiche la ligne avec le nom puis le prix. À chaque fois que la méthode **next** est invoquée, la ligne suivante devient la ligne courante, et la boucle continue jusqu'à ce qu'il n'y ai plus de ligne dans le resultset.

```
String requete= "SELECT Nom_boisson, prix FROM boisson";
ResultSet resultat = stmt.executeQuery(requete);
while(resultat.next()){
String str = resultat.getString("Nom_boisson");
float prix = resultat.getFloat("prix");
System.out.println(str + "   " + prix);
                    }
```

Le programme affichera donc

Cafe 3.2
Limonade 4.1

On peut sélectionner aussi que ce qui nous intéresse :

*" SELECT * FROM boisson WHERE Nom_boisson ='limonade'"*

La base de données va vous répondre en fournissant tous les éléments suivants :

Limonade F8 4.1

Grâce à la méthode **getXXX de JDBC**, vous pourrez alors accéder aux différents types de données SQL. Par exemple, la méthode **getInt** peut être utilisée pour accéder aux types numériques, ou caractères. Mais il est recommandé de n'utiliser **getInt** que pour accéder à des données SQL de type INTEGER

4.5. Effacement d'une ou plusieurs entrées dans la table

Pour effacer toutes les entrées d'une table

```
stmt.executeUpdate("DELETE FROM boisson");
```

Pour effacer une seule entrée dans une table:

```
stmt.executeUpdate("DELETE FROM boisson WHERE Nom_boisson ='limonade'");
```

L'entrée limonade a été effacée.

4.6. Mise à jour d'une entrée dans la table

On veut mettre à jour la colonne prix de la table boisson pour la limonade. L'instruction SQL pour mettre à jour est la suivante :

String updateString = "UPDATE boisson" + "SET prix = 2.95" +"WHERE Nom_boisson='limonade'";

Utilisant l'instruction stmt, le code JDBC exécute l'instruction SQL contenue dans updateString :

```
stmt.executeUpdate(updateString);
```

La nouvelle table est la suivante :

Nom_boisson	identifiant	prix
cafe	F12	3.2
limonade	F8	2.95

4.7. Déconnexion d'une base de donnée

L'instruction est très simple en JAVA :

```
Conn.close();
```

Ressource sur le système AIS (Automatic Identification System)

1. Présentation

Il s'agit d'un système automatique d'échange permanent de messages radio VHF entre navires.
Ce système est destiné à aider la navigation principalement dans les eaux à forte affluence (Manche par exemple)
Les transpondeurs AIS sont les équipements permettant d'émettre des messages AIS.
Actuellement, les transpondeurs sont principalement présents sur les gros bateaux soumis ou non à obligation.
Le système AIS utilise deux fréquences VHF (161,975 MHz : canal maritime 87B et 162,025 MHz : canal maritime 88B) qui sont réservées au niveau mondial pour ce système.
Ces deux canaux ne sont pas sélectionnables sur une VHF standard ce qui évite tout conflit avec les liaisons voie.
Le rôle des signaux AIS est de transmettre et recevoir des informations nautiques afin d'éviter les incompréhensions entre navigateurs et d'éviter de surcharger les canaux VHF. L'AIS est basé sur une communication radio numérique dont le temps d'émission est partagé en théorie entre 75 bateaux maximum.

2. Le protocole AIS

Le protocole AIS est un protocole à deux niveaux baptisé AIVDM/AIVDO. La couche externe est une variante du NMEA183. On donne ci-dessous le format typique d'une trame AIS :

!AIVDM,1,1,,B,177KQJ5000G?tO`K>RA1wUbN0TKH,0*5C

Le champ N°1 !AIVDM identifie le type de paquet envoyé
Le champ N°2 est 1 (non décrit)
Le champ N°3 est 1 (non décrit)
Le champ N°4 est vide numéro de message dans le cas ou plusieurs messages sont envoyés
Le champ N°5 est B (AIS canal A est une information envoyée sur la porteuse à 161.975Mhz (87B); AIS canal B est une information envoyée sur la porteuse à 162.025Mhz (88B)).
Le champ 6 représente 177KQJ5000G?tO`K>RA1wUbN0TKH ce sont les données (payload)
Le champ 7 est 0, c'est le nombre de bits utilisé pour codé un caractère dans le champ N°6.
Le champ qui suit l'étoile ici **5C correspond à l'octet de checksum.**

2.1. Etude du champ N°6 : les données

Dans un message AIS chaque caractère du champ de données est codé sur 6bits, pour retrouver les 6 bits il suffit de soustraire la valeur 48 en décimal au caractère ASCII, exemple pour les caractères suivants:

Char	ASCII	Decimal	Bits
"0"	48	0	000000
"1"	49	1	000001
"2"	50	2	000010
"3"	51	3	000011
"4"	52	4	000100

Le premier caractère du champ de données indique le type de message :

01	Position Report Class A
02	Position Report Class A (Assigned schedule)
03	Position Report Class A (Response to interrogation)
04	Base Station Report
05	Static and Voyage Related Data
06	Binary Addressed Message
07	Binary Acknowledge
08	Binary Broadcast Message
09	Standard SAR Aircraft Position Report
10	UTC and Date Inquiry
11	UTC and Date Response
12	Addressed Safety Related Message
13	Safety Related Acknowledgement
14	Safety Related Broadcast Message
15	Interrogation
16	Assignment Mode Command

17	DGNSS Binary Broadcast Message
18	Standard Class B CS Position Report
19	Extended Class B Equipment Position Report
20	Data Link Management
21	Aid-to-Navigation Report
22	Channel Management
23	Group Assignment Command
24	Static Data Report
25	Single Slot Binary Message,
26	Multiple Slot Binary Message With Communications State
27	Position Report For Long-Range Applications

3. Etude d'un message de type 1

Un message de type 1 contient 168 bits et véhicule des informations de navigation

Numero du bit	Longueur en bits	Description	Member	T	Units
0-5	6	Message Type	type	u	Constant: 1-3
6-7	2	Repeat Indicator	repeat	u	Message repeat count
8-37	30	MMSI	mmsi	u	9 decimal digits
38-41	4	Navigation Status	status	e	See "Navigation Status"
42-49	8	Rate of Turn (ROT)	turn	I3	
50-59	10	Speed Over Ground (SOG)	speed	U1	
60-60	1	Position Accuracy	accuracy	b	
61-88	28	Longitude	lon	I4	Minutes/10000 (see below)

Field	Len	Description	Member	T	Units
89-115	27	Latitude	lat	I4	Minutes/10000 (see below)
116-127	12	Course Over Ground (COG)	course	U1	Relative to true north, to 0.1 degree precision
128-136	9	True Heading (HDG)	heading	u	0 to 359 degrees, 511 = not available.
137-142	6	Time Stamp	second	u	Second of UTC timestamp
143-144	2	Maneuver Indicator	maneuver	e	See "Maneuver Indicator"
145-147	3	Spare		x	Not used
148-148	1	RAIM flag	raim	b	See below
149-167	19	Radio status	radio	u	See below