

CHAPITRE II

DES CONCEPTS AU LANGAGE

A l'issue de ce chapitre II, vous saurez :

- Donner une définition à la notion de **package** et expliquer son intérêt dans un projet Objet,
- Comment **déclarer** un package et **y faire référence** par la suite,
- Comment définir une **classe** et déterminer sa visibilité dans un projet,
- Comment construire une **méthode** et déterminer son accessibilité,
- Comment déclarer ses **variables**, après en avoir repéré les différentes catégories, et définir leur portée,
- Comment **instancier un objet** et expliquer comment se passe son implémentation en **mémoire**.

Et vous aurez fait vos premiers pas avec la notion de **polymorphisme** !

SOMMAIRE

1.	LES BASES DU LANGAGE	7
1.1.	LES COMMENTAIRES	7
1.2.	LES EXPRESSIONS	7
1.3.	LES BLOCS D'INSTRUCTIONS	7
1.4.	LES IDENTIFICATEURS	7
2.	PACKAGES	8
2.1.	INTERET	8
2.2.	DECLARATION.....	8
2.3.	IMPORTATION	9
3.	CLASSES.....	10
3.1.	INTERET	10
3.2.	DECLARATION.....	10
3.3.	MODIFICATEURS.....	11
4.	METHODES/CONSTRUCTEURS	12
4.1.	INTERET	12
4.2.	DECLARATION.....	12
4.3.	SIGNATURE	13
4.4.	MODIFICATEURS.....	13
5.	VARIABLES	14
5.1.	DECLARATION.....	14
5.2.	ATTRIBUT D'INSTANCE	15
5.3.	ATTRIBUT DE CLASSE	15
5.4.	VARIABLE DE TRAVAIL	15
5.5.	DECLARATION.....	16
5.6.	MODIFICATEURS.....	17
5.7.	EXEMPLE DE DECLARATION DE VARIABLES	18
6.	OBJETS	19

6.1.	CREATION	19
6.2.	MANIPULATION	19
7.	EXEMPLES	20
7.1.	DEFINITION DE LA CLASSE CLIENT	20
7.1.1.	Explications	20
7.1.2.	Diagramme UML	20
7.1.3.	Code java	21
7.2.	DEFINITION DU TEST CLIENT	22
7.3.	ENCAPSULATION	23
7.3.1.	Définition du Client	23
7.3.1.1.	Diagramme UML	23
7.3.1.2.	Code java	24
7.3.2.	Définition du Test Client	25
7.3.3.	Représentation Mémoire	26
7.3.4.	Représentation Mémoire d'une variable de classe	27

1. LES BASES DU LANGAGE

1.1. LES COMMENTAIRES

Un commentaire sur une ligne se déclare avec double slash : `//` .

Un commentaire sur plusieurs lignes se déclare en début avec slash étoile : `/*` et en fin avec étoile slash `*/` .

Un commentaire « javadoc », outil qui permet de générer automatiquement la documentation des applications , se déclare en début avec `/**` et en fin avec `*/` .

1.2. LES EXPRESSIONS

Une expression se termine toujours par `;` .

1.3. LES BLOCS D'INSTRUCTIONS

Un bloc permet de regrouper plusieurs instructions, il se déclare par `{` et se termine par `}` .

Toute variable déclarée dans un bloc n'est connue que de ce bloc.

1.4. LES IDENTIFICATEURS

Attention : java différencie majuscules et minuscules : « Exemple » et « exemple » sont pour lui deux identificateurs différents.

`variable` permet de stocker une donnée

- une classe commence toujours par une majuscule
 - une instance ou une variable toujours écrit en minuscule
 - intercapitalisation pour déterminer une variable (ex : `noCli`)
- ne pas hésiter à mettre des longs noms de variable

2. PACKAGES = sous-système
joue sur la visibilité, 2 classes dans deux packages
différents ne se voient pas forcément

2.1. INTERET

Un package permet de regrouper des classes et interfaces concernant un même domaine. Il s'agit d'un répertoire qui contient toutes les définitions.

La création d'un package est liée à un fichier source de définition. Ce source doit contenir en première instruction la déclaration du package auquel il appartient.

Le code source a une extension *.java*. La compilation donne lieu à une extension *.class*.
dans 98% des cas ==> 1 classe=fichier.java

Java propose différents packages.

deux classes peuvent porter le même nom tant qu'elles sont dans
des packages différents

2.2. DECLARATION

La déclaration d'un package est la première ligne source d'une classe ou interface.

Par convention un nom de package débute en minuscule, et ne doit pas contenir de caractère blanc.

```
package nom_du_package
```


2.3. IMPORTATION

Par défaut, l'utilisation d'une classe n'est possible qu'à l'intérieur d'un même package. Quand une classe doit utiliser une classe définie dans un autre package, elle est obligée de donner le chemin complet de l'endroit où se trouve la classe ou plus simplement importer le package auquel l'autre classe appartient.

Il existe deux types d'importation :

- L'importation de l'ensemble d'un package.
- L'importation d'un seul élément d'un package.

```
package nom_du_package1 ;
class ClasseA { ... }
class ClasseB { ... }

// Exemple sans importation
package nom_du_package2 ;
class ClasseC {
    nom_du_package1.ClasseA ...
}

// Exemple avec importation totale
package nom_du_package2 ;
import nom_du_package1.* ;
class ClasseC {
    ClasseA ...
    ClasseB ...
}

// Exemple avec importation d'une classe
package nom_du_package2 ;
import nom_du_package1.ClasseA ;
class ClasseC {
    ClasseA ...
    nom_du_package1.ClasseB
}
```

Remarque : Les classes du package `java.lang.*` sont importées par défaut.

3. CLASSES

3.1. INTERET

Une classe définit un type, utilisable ensuite comme un type de données.

3.2. DECLARATION

La déclaration s'effectue par le mot clé *class* et l'on indique le nom de la classe.

Par convention un nom de classe débute en majuscule, et ne doit pas contenir de caractère blanc. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

Devant le mot clé, on peut trouver des modificateurs. Derrière on note l'héritage par les mots clés :

- *extends* pour hériter d'une classe.
- *implements* pour les interfaces.

```
class ClasseA {...}
```

Exemple avec héritage

```
class ClasseB extends ClasseA{...}
```

Exemple avec modificateurs

```
public class ClasseC {...}
```

Syntaxe

```
[Modificateurs] class NomClasse [extends NomSuperClasse] [implements Interface1, Interface2] {...}
```

3.3. MODIFICATEURS

La visibilité d'une classe est effectuée par les modificateurs. Par défaut une classe est visible à l'intérieur d'un même package.

- **public** : la classe est visible pour les autres packages.
- **final** : la classe ne pourra pas être sur-classe.
- **abstract** : la classe ne pourra pas être instanciée, elle sert de référence à des sous-classes.

4. METHODES/CONSTRUCTEURS

4.1. INTERET

Une **méthode** définit un traitement à effectuer.

Le **constructeur** est une méthode particulière de la classe, il permet l'initialisation des variables.

4.2. DECLARATION

La déclaration peut commencer par un modificateur, qui indique le degré de visibilité.

Une méthode retourne toujours une information ou *void* s'il n'y en a pas.

Entre parenthèse on indique les paramètres reçus (séparés par des virgules).

Par convention un nom de méthode débute en minuscule, et ne doit pas contenir de caractère blanc. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

Un constructeur porte le même nom que la classe, le constructeur se déclare au début, et ne retourne rien à l'appelant donc pas de *void* à noter.

```
class ClasseA {  
  // Constructeur  
  ClasseA() {...}  
  // Autres méthodes  
  void neRetourneRien() {...}  
  boolean retourneBooleen() { return false ;}  
  void recoitBooleen (boolean a) {...}  
  // Syntaxe  
  [Modificateurs] type de retour nomMethode([typeArg1 nomArg1, typeArg2 nomArg2]) {...}
```

4.3. SIGNATURE ==> plusieurs méthodes qui portent le même nom mais doivent avoir des signatures différentes

Un nom de méthode n'est pas unique. Une même classe peut avoir plusieurs constructeurs et plusieurs fois le même nom de méthode.

L'association :

- du nom de méthode
- du nombre de paramètres reçus et de leur type

... constitue la **signature** de la méthode. Cette signature est **unique**.

Quand une classe contient des méthodes ayant le même nom, on parle de **surcharge**.

On parle de **redéfinition** quand on redéfinit dans une sous-classe une méthode de la classe-mère (cf. l'héritage chapitre III).

4.4. MODIFICATEURS

La visibilité d'une méthode est déterminée par les modificateurs.

Par défaut une méthode est visible à l'ensemble du package.

- **private** : méthode visible par la classe uniquement.
- **public** : méthode visible par tous les packages.
- **protected** : méthode visible par les sous-classes.
- **static** la méthode pourra être invoquée pour la classe elle-même (sans avoir besoin d'instancier un objet)

5. VARIABLES

5.1. DECLARATION

Chaque variable ou donnée doit être déclarée avant d’être manipulée.

La déclaration d’une variable doit toujours être précédée du type de variable que l’on veut manipuler.

Par convention un nom de variable débute en minuscule, et ne doit pas contenir de caractère blanc. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

On peut déclarer plusieurs variables en même temps, dans ce cas on les sépare par des virgules.

On peut initialiser une variable en même temps que sa déclaration.

Les types utilisés pour le moment sont :

Type		
boolean	Booléen valant <i>true</i> ou <i>false</i>	
char	Un seul caractère, à définir avec de simples quotes : 'e' utilisé pour les sexes et V/F	
int	Numérique entier	
float	double	Numérique avec décimales
String	Chaîne de caractères, à définir avec des doubles quotes : "e"	

il s'agit d'une classe car S Si 1 seul caractère utilisé char car avec String : il y a le caractère + symbole qui indique que la chaine est finie donc 2 places au lieu d'une avec char

// **Déclaration d'une variable**

```
// Déclaration d'une variable
int nomVarEntier;

// Déclaration de plusieurs variables
int nomVarEntier1, nomVarEntier2, nomVarEntier3 ;

// Déclaration et initialisation
int nomVarEntierN = 5 ;
```

Syntaxe :
[Modificateurs] type de la variable nomVariable

5.2. ATTRIBUT D'INSTANCE

Un attribut d'instance (ou « variable d'instance ») est une donnée définie en dehors de toute méthode. Chaque objet possèdera sa propre valeur : par exemple, le nom d'une personne.

C'est une variable globale à l'ensemble des méthodes de la classe : toutes peuvent y accéder.

5.3. ATTRIBUT DE CLASSE

Une classe peut posséder un attribut qui sera partagé par toutes ses instances (par exemple le nombre d'instances créées).

Une « variable de classe » est créé lors du chargement de la classe en mémoire vive (et non lors de l'instanciation d'objets). Elle doit être créée avec le modificateur *static*.

Une donnée définie avec le modificateur *final* est une constante. Elle mérite donc d'être également *static*.

5.4. VARIABLE DE TRAVAIL

Une méthode peut déclarer une variable, dans ce cas elle reste locale à la méthode et n'est pas visible par d'autres méthodes.

Une donnée définie avec le modificateur *final* est une constante.

5.5. DECLARATION

Toute donnée, ou variable doit être déclarée avant d'être manipulée.

La déclaration d'une donnée doit toujours être précédée du type de donnée que l'on veut manipuler.

Par convention un nom de donnée débute par une minuscule, et ne doit pas contenir de caractère spécial. Si le nom est composé de plusieurs mots on met une majuscule à chaque nouveau mot.

On peut déclarer plusieurs données en même temps, dans ce cas on les sépare par des virgules.

On peut initialiser une donnée en même temps que sa déclaration.

// Déclaration d'une variable

```
int nomVarEntier;
```

// Déclaration de plusieurs variables

```
int nomVarEntier1, nomVarEntier2, nomVarEntier3 ;
```

// Déclaration et initialisation

```
int nomVarEntiern=5 ;
```


// Syntaxe :

```
[Modificateurs] type de la variable nomVariable [=valeur];
```


5.6. MODIFICATEURS

La visibilité des attributs (d'instance ou de classe) est déterminée par les modificateurs.

Par défaut un attribut est visible à l'ensemble du package.

- **private :**  obligatoire si on veut avoir l'encapsulation attribut visible par la classe uniquement.
- **public :** attribut visible par tous les packages.
- **protected :** attribut visible par les sous-classes.
- **final** déclaration d'une constante.
- **static** attribut partagé par toutes les instances.

Remarque : Une variable de travail n'est accessible que dans la méthode où elle a été déclarée. La notion de modificateur ne s'applique donc pas, sauf éventuellement *final* s'il s'agit d'une constante.

5.7. EXEMPLE DE DECLARATION DE VARIABLES

```

class ClasseA
{
// Variable d'instance
int nomVarEntier;
// Constante partagée par toutes les instances
static final double PI = 3.141 ;
// Constructeur : initialisation des variables d'instance
ClasseA()
{
    nomVarEntier = 10 ;
}
// Autres méthodes
void neRetourneRien()
{
    // Variable de travail
    boolean nomVarBool = false ;
}
}

```

Les constantes sont déclarées en majuscules pour les différencier du reste
Si plusieurs mots : NB_ELEMENTS

un constructeur n'est pas indispensable
Le constructeur reprend le nom de la classe
S'il y en a pas de déclaré, il y en a un par défaut

6. OBJETS

6.1. CREATION

La création d'un objet s'effectue avec l'opérateur *new*, puis on utilise le nom du constructeur de la classe que l'on veut instancier.

```
// Instanciation
```

```
new ClasseA() ;
```

6.2. MANIPULATION

Pour manipuler cet objet, il doit être mémorisé dans une variable. Le type de la variable est la classe.

```
// Définition de la Classe
```

```
public class ClasseA  
{  
    // Constructeur  
    public ClasseA { }  
    // Méthode  
    public int traitement() {return 100;}  
}
```

```
// Définition des variables de travail
```

```
ClasseA variable;
```

```
int retour;
```

```
// Instanciation
```

```
variable = new ClasseA() ;
```

```
retour = variable.traitement();
```

```
ClasseA autre = new ClasseA();
```

```
int test = autre.traitement();
```

7. EXEMPLES

7.1. DEFINITION DE LA CLASSE CLIENT

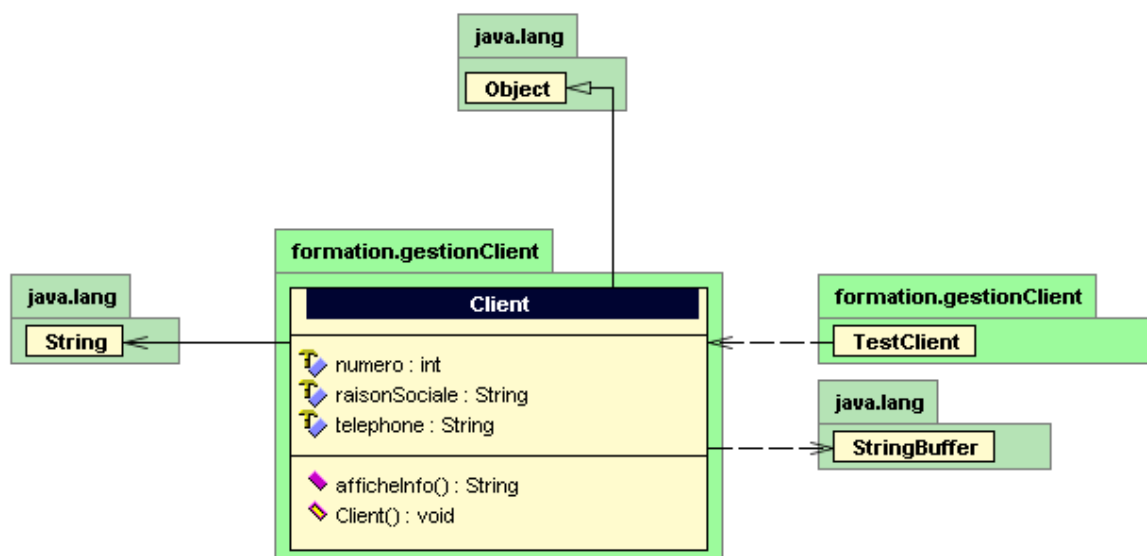
7.1.1. EXPLICATIONS

Pour cette classe Client, on mémorise le numéro du client, sa raison sociale et son numéro de téléphone. On définit :

- un constructeur, recevant en argument le numéro de client à créer.
- Un traitement permettant de retourner une chaîne de caractères affichant le numéro et la raison sociale du client, par exemple « le client est : 123 ETABLISSEMENT DUPONT ». L'opérateur de concaténation est le signe +.

Cette méthode se nomme afficheInfo().

7.1.2. DIAGRAMME UML



7.1.3. CODE JAVA

```
package formation.gestionClient ;  
  
public class Client  
{  
    // Déclaration des variables d'instance  
    int numero ;  
    String raisonSociale ;  
    String telephone ;  
  
    // Constructeur : permet d'initialiser les variables d'instance  
    public Client (int num)  
    {  
        numero = num ;  
    }  
  
    // Envoi d'une chaine de caractères affichant le numéro et la raison sociale.  
    public String afficheInfo()  
    {  
        return "le client est : " + numero + " " + raisonSociale ;  
    }  
}
```

7.2. DEFINITION DU TEST CLIENT

Un programme exécutable en java doit posséder une méthode *main*. Cette méthode ne retourne aucun argument et doit être déclarée avec les modificateurs *public* et *static*. Elle reçoit en paramètre un tableau de chaîne de caractères.

Pour afficher des informations sur la console système, on utilise la méthode *println* de la classe *System* pour la variable de classe *out*.

```
class TestClient
{
    // Déclaration du constructeur
    TestClient ()
    {}

    // Programme principal
    public static void main (String args[])
    {
        // création d'un objet client
        Client client1 = new Client (1000);
        // modification de la raison sociale de l'objet client1
        client1.raisonSociale = "Société A";
        // modification du téléphone de l'objet client1
        client1.telephone = "0158225858";
        // affichage à la console système du traitement d'affichage du client1
        System.out.println( "AFFICHAGE " + client1.afficheInfo() );
    }
}
```

Résultat obtenu à la console système :

```
AFFICHAGE le client est : 1000  Société A
```

7.3. ENCAPSULATION

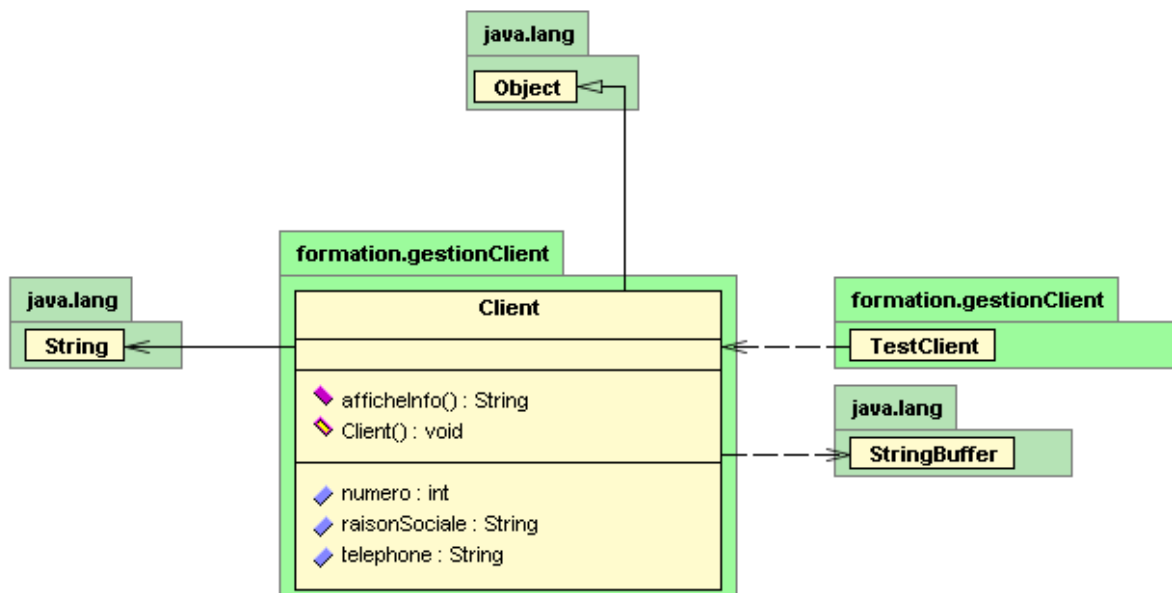
7.3.1. DEFINITION DU CLIENT

Modification de la classe Client, afin de mettre en œuvre l'encapsulation.

On rend les variables d'instance inutilisables, en mettant le modificateur *private*.

On définit les accesseurs, permettant de récupérer et de mettre à jour les variables d'instances.

7.3.1.1. Diagramme UML



7.3.1.2. Code java

```
package formation.gestionClient ;

public class Client
{
    // Déclaration des variables d'instance
    private int numero ;
    private String raisonSociale ;
    private String telephone ;

    // Constructeur : permet d'initialiser les variables d'instance

    public Client (int num) {
        numero = num ;
    }

    // Accesseurs

    public String getRaisonSociale() {
        return raisonSociale ;
    }
    public void setRaisonSociale(String raison) {
        raisonSociale = raison ;
    }
    public String getTelephone() {
        return telephone;
    }
    public void setTelephone(String phone) {
        telephone = phone ;
    }
    public int getNumero() {
        return numero ;
    }
    public void setNumero(int num) {
        numero=num;
    }

    // Envoi d'une chaine de caractères affichant le numéro et la raison sociale.
    public String afficheInfo() {
        return « le client est » + numero + « »+ raisonSociale ;
    }
}
```


7.3.2. DEFINITION DU TEST CLIENT

Le client test ne peut plus modifier directement la raison sociale et le numéro de téléphone. Il doit donc manipuler les accesseurs.

```
class TestClient
{
    // Déclaration du constructeur
    TestClient ()
    {}

    // Programme principal
    public static void main (String args[])
    {
        // Test d'un premier objet client
        // Création d'un premier objet client
        Client client1 = new Client (1000);
        // Modification de la raison sociale de l'objet client1
        client1.setRaisonSociale ("Société A");
        // Modification du téléphone de l'objet client1
        client1.setTelephone("0158225858");
        // Affichage à la console système du traitement d'affichage du client1
        System.out.println( "AFFICHAGE " + client1.afficheInfo() );
        // Affichage à la console système du téléphone du client1
        System.out.println( "AFFICHAGE " + client1.getTelephone() );

        // Test d'un deuxième objet client
        // Création d'un deuxième objet client
        Client client2 = new Client (2000);
        // Modification de la raison sociale de l'objet client2
        client2.setRaisonSociale ("Etablissement D");
        // Affichage à la console système du traitement d'affichage du client2
        System.out.println( "AFFICHAGE " + client2.afficheInfo() );
    }
}
```

Résultat obtenu à la console système :

```
AFFICHAGE le client est : 1000  Société A
AFFICHAGE 0158225858
AFFICHAGE le client est : 2000  Etablissement D
```

7.3.3. REPRESENTATION MEMOIRE

A l'exécution du client test, un objet de type ClientTest a été créée par la machine virtuelle. La méthode *main* de cet objet n'est présente qu'une seule fois en mémoire car elle est déclarée en *static*.

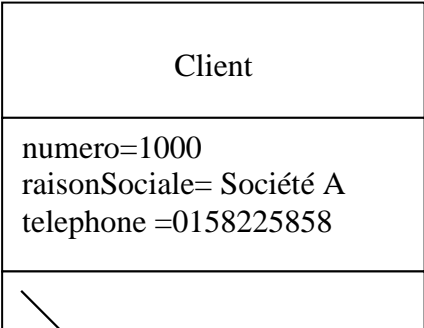
Par contre ClientTest lance la création de 2 objets client1 et client2, les variables d'instances de ces objets sont présents en mémoire pour chacune des instances, les méthodes sont chargées à partir de la définition de la classe.

Mémoire :

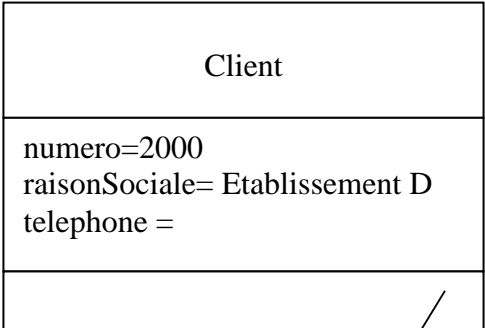
Les objets



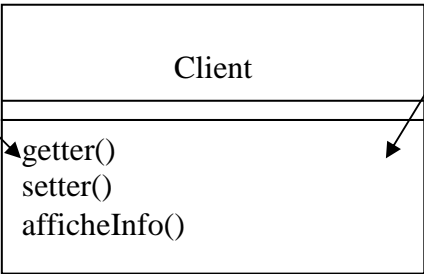
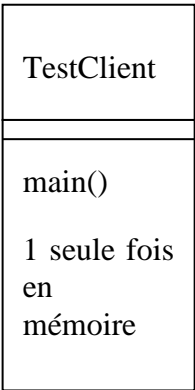
client1



client2



Les classes



7.3.4. REPRESENTATION MEMOIRE D'UNE VARIABLE DE CLASSE

Dans la définition du client on ajoute une variable de classe :

Par exemple, un compteur permettant de connaître le nombre d'objets client créé.

```
package formation.gestionClient ;
public class Client
{
    // Déclaration des variables d'instance
    private int numero ;
    private String raisonSociale ;
    private String telephone ;
    private static int cpt ;
    // Constructeur : permet d'initialiser les variables d'instance

    public Client (int num)
    {
        numero = num ;
        cpt = cpt + 1 ;
    }
}
```

Mémoire :

Objet

client1

Client
numero=1000 raisonSociale= Société A telephone =0158225858

client2

Client
numero=2000 raisonSociale= Etablissement D telephone =

Classe

Client
cpt = 2
getter() setter() afficheInfo()

