Quelle est la différence entre une CLASSE et un OBJET

Une **classe** est le « blueprint » pour un **objet.** Il dit à la machine virtuelle comment faire un objet d'un type particulier. Chaque objet fait à partir de cette classe peut avoir ces propres valeurs pour les variables d'instance de cette classe.

Quand vous aller designer une classe, penser à propose des objets qui seront crées à partir de cette classe. Penser à :

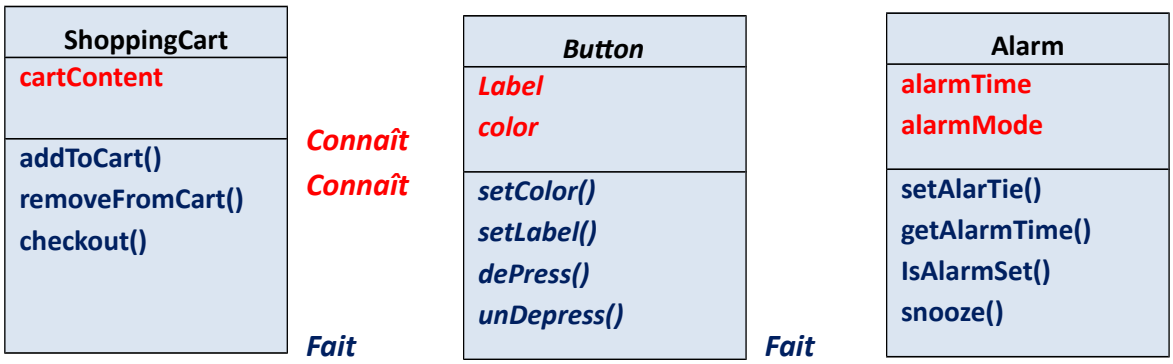
- **Les choses que l'objet connaît**
- **Les choses que l'objet fait**

Les choses que connaît un objet sont appelées :

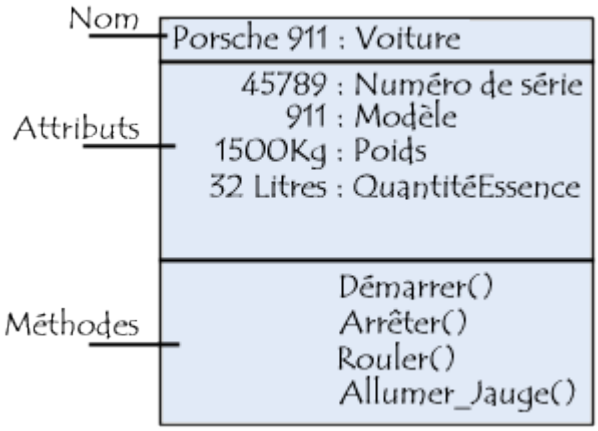
- **Les variables d'instance**

Les choses que fait un objet sont appelées :

- **Les méthodes**



Par exemple vous pouvez utiliser la classe « Button » pour faire de douzaines de différents buttons, et chaque button peut avoir sa propre couleur, grandeur, forme, marque, et plus.



EXEMPLE 1

FAIRE VOTRE PREMIER OBJET

1. Ecrire votre CLASSE

```
Class Dog {
    Int   size ;
    String  breed ;
    String name ;
    Void bark () {
        System.out.println (« Ruff ! Ruff ! ») ;
        }
    }
```

Les variables d'instance

Une méthode

Dog
size breed name
bark()

2. Ecrire votre tester (TestDrive) class

```
class DogTestDrive  {
    Public static  main (String  [ ]  args)  {
        //Dog  test  code goes  here
            }
    }
```

3. Dans votre tester, fait un objet et accède les variables et les méthodes de l'objet

```
Class  DogTetDrive  {

        public  static   void  main  (String [ ]  args )  {

                Dog   d  =  new  Dog();

                d.size  = 40;

                d.bark ();

                }

        }
```

Fait un objet dog

L'opérateur dot

L'opérateur « Dot » (.)

L'opérateur dot (.) vous donne l'accès à l'état et le comportement (Les variables d'instances et de méthodes).

```
//Fait un nouveau objet

Dog   d = new  Dog () ;

//Dis-le d'abouayer (to bark)
//en utilisant l'opérateur
dot //sur la variable  d
pour //appeler « to bark() »

d.bark() ;

//fixer sa grandeur en
//utilisant l'opérateur dot

d.size  = 40 ;
```

ENCAPSULATION

OOP cache les données (attributs) et les fonctions (comportement) en parquets objets; les données et les fonctions d'un objet sont attachées ensemble. Les objets ont la propriété de cacher les informations. Cela signifie que, pendant les objets peuvent connaitre comment communiqué avec un autre à travers les interfaces définies, ils ne sont pas permit de connaitre comment sont implémenter les autres objets. Les détails des implémentations sont caches dans les objets eux-mêmes. Sûrement il est possible de conduire une voiture sans connaitre les détails comment le moteur, la transmission et les autres parties marchent intérieurement.

- L'encapsulation consiste à masquer les détails d'implémentation d'un objet, en définissant une interface.

- L'interface est la vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.
- L'encapsulation facilite l'évolution d'une application car elle stabilise l'utilisation des objets : on peut modifier l'implémentation des attributs d'un objet sans modifier son interface.
- L'encapsulation garantit l'intégrité des données, car elle permet d'interdire l'accès direct aux attributs des objets (utilisation d'accesseurs).

Exposer nos données pour que quelqu'un d'autre les voies et même les touchées.
Exposer veut dire joignable avec l'opérateur dot (point) comme ceci :

LeChat.hauteur = 27 ;

LeChat.hauteur = 0 ; Nous n'allons pas permettre ça. Par conséquent, nous devrons construire une la méthode « setter » pour toutes les variables, et trouver un moyen qui forcera les autres codes d'appeler les « setters » plutôt que d'accéder les données directement.

Comment cachée les données ?

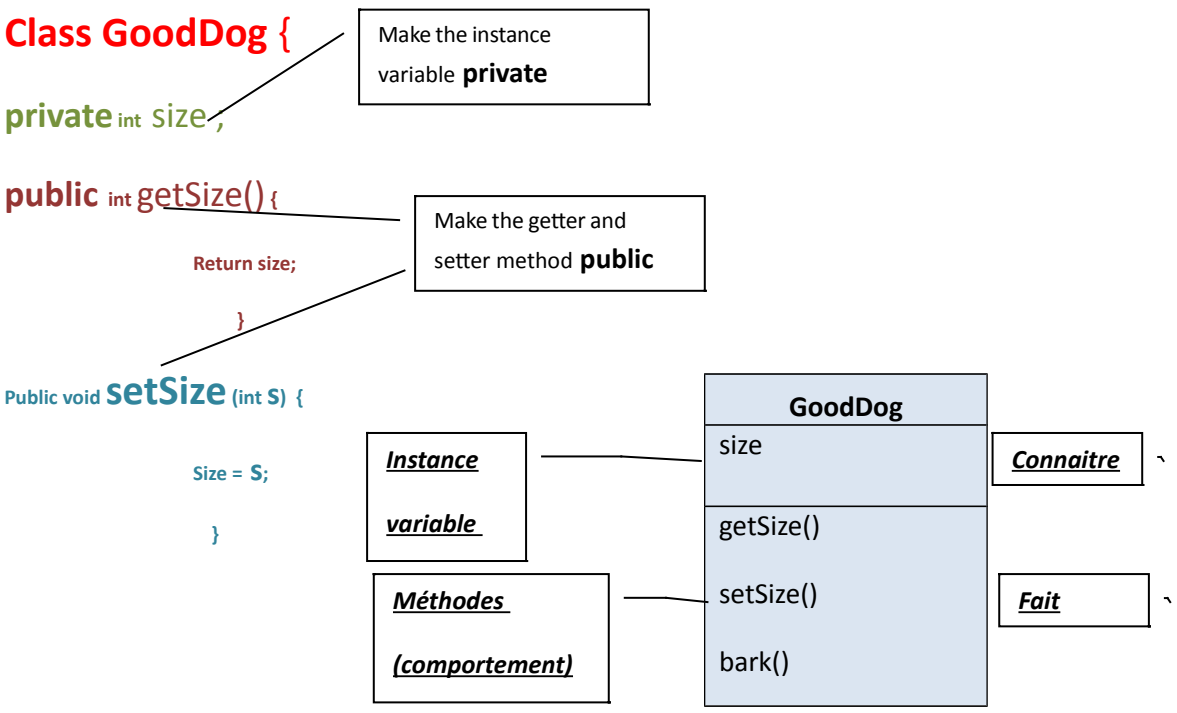
Avec le PUBLIC et le PRIVATE (access modifiers). Marquer vos variables instances comme **private** et donner au **public** les «setters » et les « getters » pour le control d'accès.

- ✓ Marquer les variables instance **private.**
- ✓ Marque les getters et setters **public.**

Protected

Le modifier **Protected** sert comme un niveau intermédiaire de protection d'accès entre le **Public** et le **Private.**

EXEMPLE 2



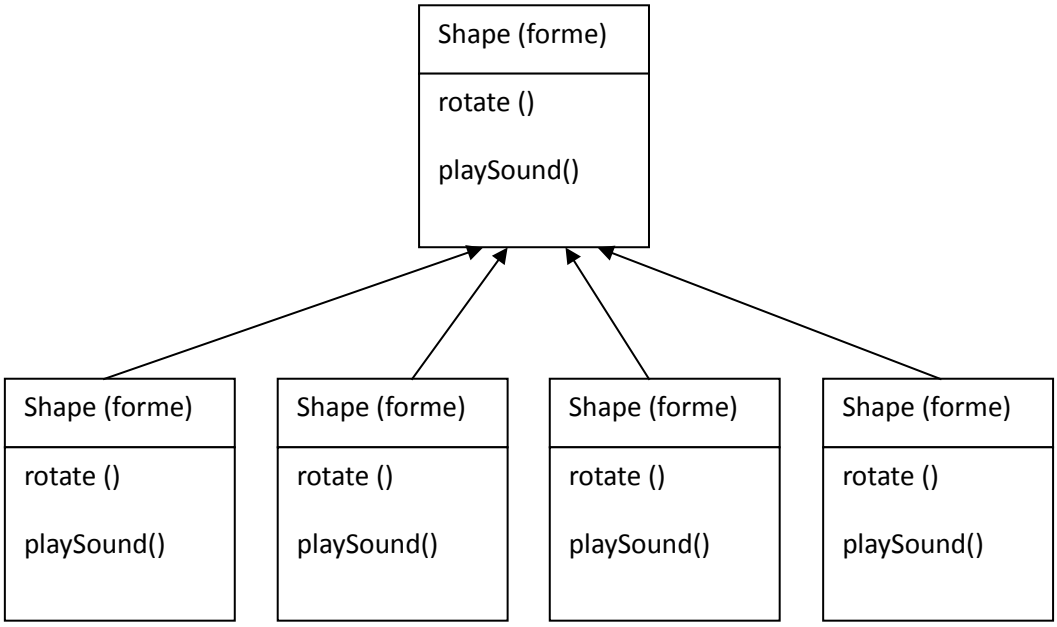
```
                    }
Void bark () {
if (size > 60) {
                    System.out.println("Woof! Woof!");
      } else if (size > 14) {
                    System.out.println("Ruff! Ruff!");
          } else {
                    System.out.println ("Yip!  Yip!);
}
}
Class  GoodDogTestDrive {
          Public static void main (String [] args) {
          One.setSize (70);
          GoodDog   two = new   GoodDog();
          Two.setSize (8);
          System.out.println ("Dog  one: " +  one.getSize());
          System.out.println  ("Dog  two:" +  two.getSize());
          One.bark  ();
          Two.bark ();
}
}
```

EXEMPLE 3

---------- Héritage (inheritance)

Il y'a de beaucoup de formes tels que rectangle, carré, cercle…. Elles roulent et en procurent des sons (playsound).

Ici on va soutirer certains renseignements et les mettre dans une nouvelle classe appelée Shape (forme).

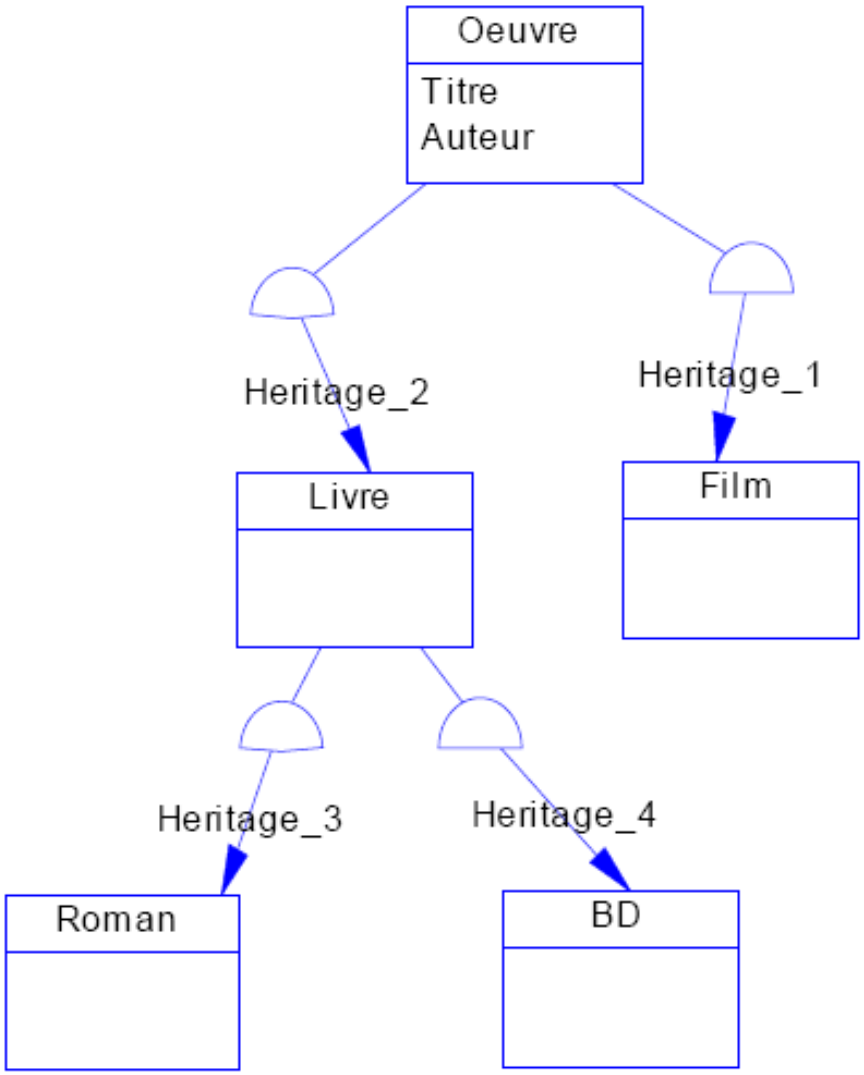


Cacher les informations est très critique pour un bon logiciel.

OOAD = Object Oriented Analysis and Design process.

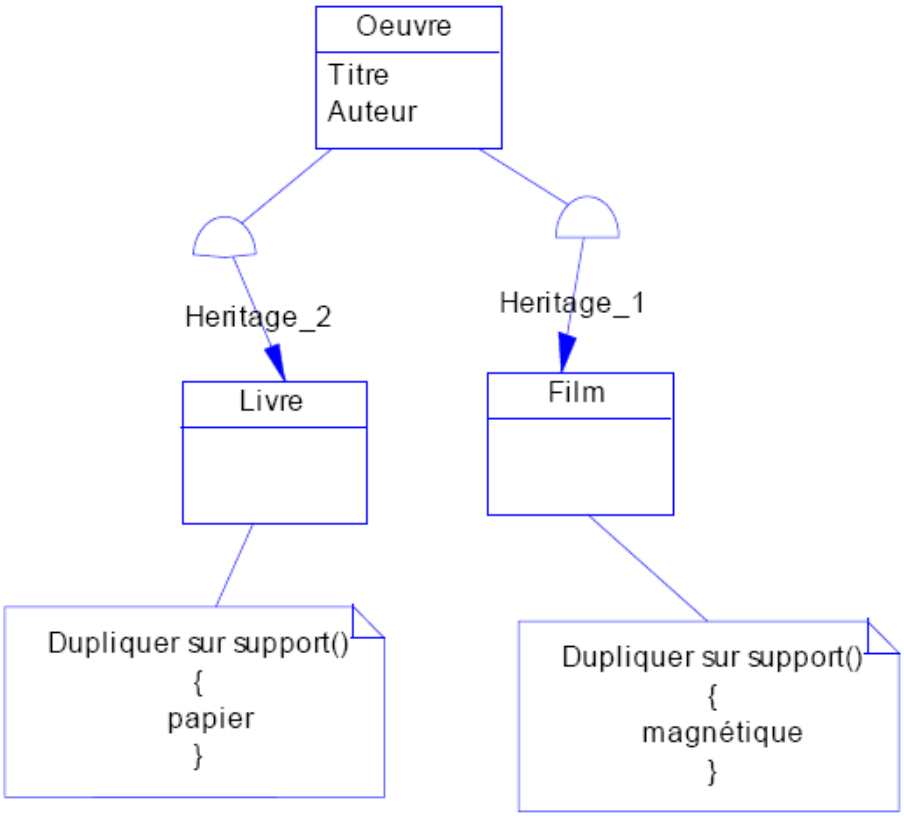
HERITAGE

- L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe.
- Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines.
- Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes.
- La spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple.
- L'héritage évite la duplication et encourage la réutilisation.
- Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes.
- Le polymorphisme augmente la généricité du code.



POLYMORPHISME

- Le polymorphisme représente la faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes (Le **polymorphisme** représente la faculté d'une même opération de s'exécuter différemment suivant le contexte de la classe où elle se trouve).
- Le polymorphisme augmente la généricité du code.
- Ainsi, une opération définie dans une superclasse peut s'exécuter de façon différente selon la sous-classe où elle est héritée.
- Ex : exécution d'une opération de calcul des salaires dans 2 sous-classes spécialisées : une pour les cadres, l'autre pour les non-cadres.



- **l'agrégation**

Il s'agit d'une relation entre deux classes, spécifiant que les objets d'une classe sont des composants de l'autre classe.
Une relation d'agrégation permet donc de définir des objets composés d'autres objets.
L'agrégation permet d'assembler des objets de base, afin de construire des objets plus complexes.