Fiche 1
Encapsulation

En POO le terme **Encapsulation** fait référence à **deux notions** distinctes. Il est utilisé pour désigner l'une des deux, et parfois à la combinaison de celles-ci :

- Un mécanisme du langage pour **restreindre** l'accès à certains composants de l'objet
- Une structure du langage, qui facilite le **regroupement** des données avec les méthodes (ou fonctions) d'exploitation de ces données.

Les langages tels que C++, Java, Ruby ou Python, Php et plus généralement tous les langages influencés par la syntaxe du C++, proposent au moins trois niveaux de visibilité :

- **public** : les attributs dits publics sont accessibles à tous,
- **protégé** : les attributs dits protégés sont accessibles seulement aux classes dérivées,
- **privé** : les attributs privés sont accessibles seulement par l'objet lui-même.

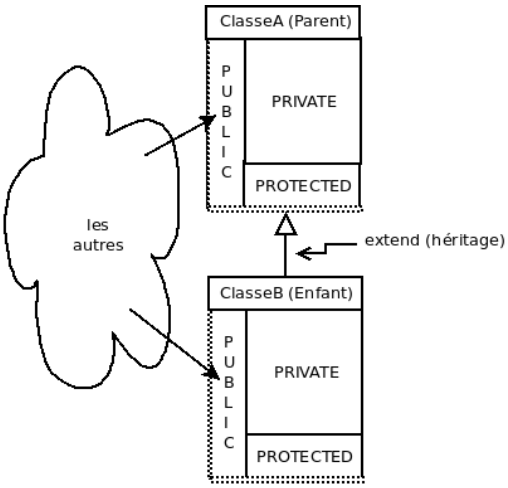
Attention, présenté ainsi dans ce document (et dans les langages courants), ce critère d'encapsulation ne permet pas la distinction entre la lecture et l'écriture d'un attribut.

Les principaux critères de restriction généralement rencontrés dans les langage de programmation sont :

Critère de restriction (niveau de visibilité)	*Notation UML*	*Java/Php*	
public	+		
privé	-		
protégé	#		

Règles de regroupement :

- Placer les données et opérations qui agissent sur ces données dans la même classe
- Utiliser la logique « qui est responsable de quoi » pour déterminer le regroupement des données et des opérations dans les classes. (voir patrons GRASP de Craig Larman)



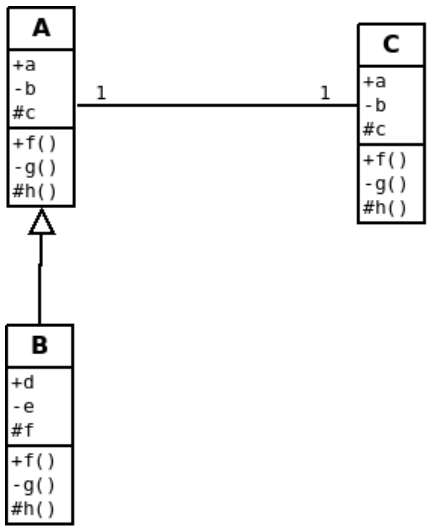
Règles de masquage :

- Ne pas directement exposer des données
- Ne pas exposer la différence entre données stockées et données dérivées. Par exemple :
  getAgeCalculé() ne cache pas le fait que l'age est déterminé par un calcul **:-(à éviter**
  getAge() : ne dit rien sur la façon dont l'âge est déterminé (attribut ou calcul) **:-) conseillé**

Principe sous-jacent : *Open-Close Principle* (https://fr.wikipedia.org/wiki/Principe_ouvert/ferm%C3%A9)

Références

Wm. Paul Rogers, *Encapsulation is not information hiding*, JavaWorld.com, 05/18/01
Principe de conception : Ouvert/Fermé
Patrons de conception : Patrons_GRASP
Wikipedia fr : Encapsulation (programmation)
Wikipedia us : Encapsulation (object-oriented_programming)



Exercices

Compléter le tableau ci-dessous constitué de questions portant sur le diagramme de classes UML ci-contre.
A et C sont reliées par un lien d'association bi-directionnel
B est une sous-classe de A

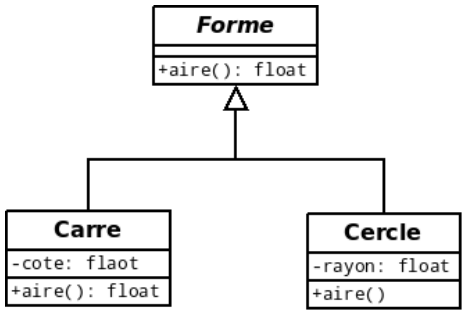
Question	Réponse	Justification
Une instance de **A** peut-elle accéder à l'attribut **a** d'un objet de type **C** ?		
Une instance de **A** peut-elle accéder à l'attribut **b** d'un objet de type **C** ?		
Une instance de **A** peut-elle accéder à l'attribut **c** d'un objet de type **C** ?		
Une instance de **A** peut-elle solliciter la méthode **f** d'un objet de type **C** ?		
Une instance de **A** peut-elle solliciter la méthode **g** d'un objet de type **C** ?		
Une instance de **A** peut-elle solliciter la méthode **h** d'un objet de type **C** ?		
Une instance de **B** peut-elle accéder à l'attribut **a** d'un objet de type **C** ?		
Une instance de **B** peut-elle accéder à l'attribut **b** d'un objet de type **C** ?		
Une instance de **B** peut-elle accéder à l'attribut **c** d'un objet de type **C** ?		
Une instance de **B** peut-elle solliciter la méthode **f** d'un objet de type **C** ?		
Une instance de **C** peut-elle solliciter la méthode **f** d'un objet de type **A** ?		
Une instance de **C** peut-elle solliciter la méthode **h** d'un objet de type **A** ?		
Une instance de **B** peut-elle solliciter la méthode **f** de sa classe mère **A** ?		
Une instance de **C** peut-elle solliciter la méthode **f** d'un objet de type **B** ?		

Fiche 2
Polymorphisme

Le terme **Polymorphisme** fait référence à la notion d'**interface** et d'**implémentation**.

C'est la « capacité d'appeler une variété de fonctions ayant exactement la même interface » [Stroustrup].

Voici un exemple :




```
abstract class Forme {
   abstract float aire() ;

}

class Carre extends Forme{
   private float cote;
   public float aire() {
       return cote * cote;
   }
}

class Cercle extends Forme{
   private float rayon;
   public float aire() {
       return Math.PI*rayon*rayon;
   }
}
```

Autres expressions de la classe parent :

```
interface Forme {
   public float aire() ;
}
```

Dans ce dernier cas, on dit que les les classes 'enfants' implémentent l'interface et on utilise le mot clé ***implements*** à la place de ***extends (fiche suivante)***

Pour illustrer le caractère polymorphe de la méthode aire, nous définissons un traitement générique qui fonctionnera avec tout objet dont la classe implémente Forme :

```
class TestForme{

   static void showAire(List<Forme> formes) {
      for(Forme f : formes)
        System.out.println("aire :" + f.aire());
   }
}
```

L'appelant ne sait pas à l'avance (au moment de la compilation) quelles sont les classes qui héritent (ou qui implémentent) Forme. Malgré le fait que l'appel soit toujours exprimé de la même façon (`f.aire()`) les algorithmes déclenchés peuvent être différents (différentes fonctions réparties dans des classes héritant directement ou non de Forme et définissant à leur manière la fonction aire) – d'où le terme poly (plusieurs) morphe (forme – représentée par l'implémentation dans notre cas).

Références

Divers définitions de polymorphisme
http://stason.org/TULARC/software/object-oriented-programming/index.html
http://stason.org/TULARC/software/object-oriented-programming/2-1-What-Is-Polymorphism-Other-Definitions.html
http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_objet#Le_typage_et_le_polymorphisme

Fiche 3
Classe et Interface : une question d'implémentation

La *classe* décrit la structure interne des **données** et définit les **méthodes** applicables aux objets de même type (classe).

Les propriétés publiques d'une classe définissent son **interface** (au sens large)

Une classe dispose toujours d'au moins une fonction particulière d'*initialisation* de ses objets : **constructeur**.

Les objets sont dits *instances de la classe*. C'est pourquoi les attributs d'un objet sont aussi appelés ***variables d'instance*** et les opérations associées des ***méthodes d'instance***.

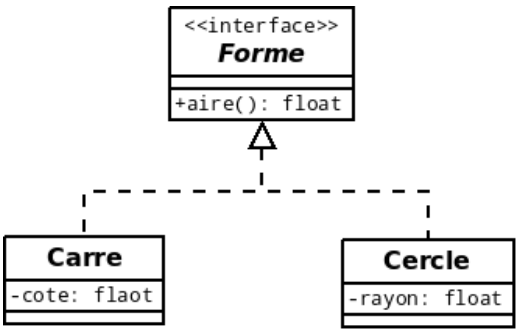
On peut définir un type d'objet sans référence à une implémentation : c'est la notion de **classe purement abstraite** ou **interface**. Les mots clés **abstract** et **interface** permettent cela dans les langages objet.

Voici un exemple :

```
interface Forme {
   float aire();
}

class Carre implements Forme{
   private float cote;
   public float aire() {
      return cote * cote;
   }
   ...
}

class Cercle implements Forme{
   private float rayon;
   public float aire() {
      return Math.PI*rayon*rayon;
   }
   ...
}
```



représentation UML

Dans cet exemple, nous avons clairement une interface et 2 classes d'implémentation.

Une interface peut hériter d'une ou plusieurs autres interfaces (java accepte l'héritage multiple, mais entre interfaces seulement).

Références

http://en.wikipedia.org/wiki/Interface_(Java)

Bruce Eckel : http://penserenjava.free.fr/

Le concept d'interface a permis d'implémenter l'*inversion de dépendance (IDP)* et son pendant, *l'inversion de contrôle (IOC).*

EXERCICES

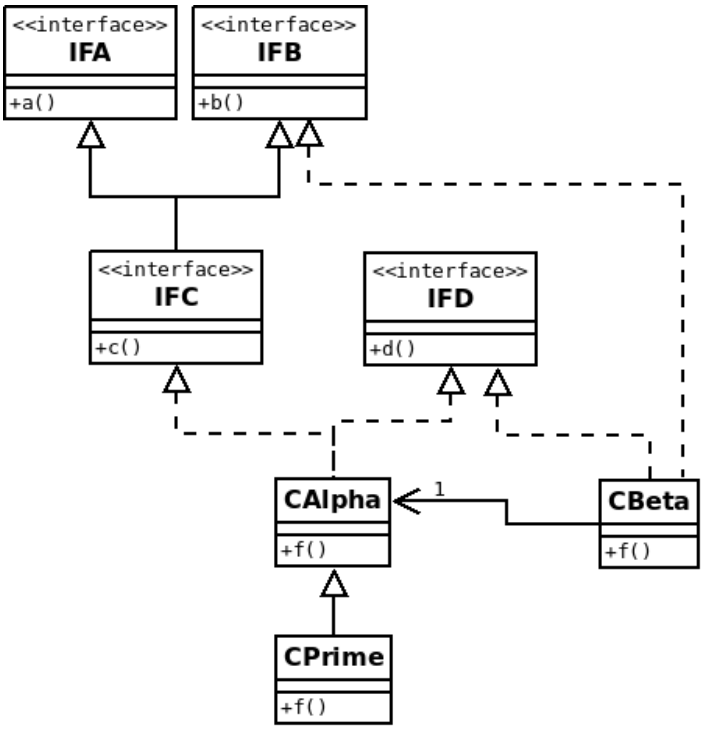
Compléter la traduction java du diagramme de classes ci-contre, étant donné que toutes les opérations sont des procédures non paramétrées, que les classes CAlpha et CBeta sont instanciables, et les rôles des instances sont triviaux.

```
interface IFA { public void a(); }

class CBeta implements IFD, IFB {
 private CAlpha alpha;
 public CBeta( CAlpha alpha ) {
     this.alpha = alpha;
 }
  public void f()  {  }
  public void d() {  }

}
```

à compléter



Questionnement

Quel est le rôle de **this** dans le corps du constructeur ?

Vérifier la **validité** des instructions suivantes (en relation au diagramme de classe de l'exercice A) :

	Classe CBeta	**Commentaire**
1	`public void f() { alpha.f(); }`	OK, l'appel à f() est compatible avec le type de l'attribut alpha.
2	`public void f() { this.f(); }`	
3	`public void f() { this.b(); this.c(); }`	
4	`public void f() { this.b(); alpha.b();}`	
5	`public void f() { b(); this.alpha.b();}`	
6	`public void f() { return; }`	
7	`public void f() { this.alpha.a(); }`	
8	`public void f() { this.d(); this.c(); }`	
	Classe CAlpha	
9	`public void f() { this.b(); }`	
10	`public void f() { this.f(); }`	
11	`public void f() { alpha.f(); }`	
12	`public void f() { alpha.a();}`	
13	`public void f() { a(); a(); this.d();}`	
14	`public void f() { beta.f(); }`	
	Instruction : déclaration + initialisation	
15	`IFC x = new CAlpha();`	
16	`IFD x = new CAlpha();`	
17	`IFA x = new CAlpha();`	
18	`CBeta x = new CAlpha();`	
19	`IFB x = new CBeta();`	
20	`IFA x = new CBeta();`	
21	`IFD x = new CBeta();`	
22	`IFB x = new CAlpha();`	
23	`CAlpha x = new CPrime();`	